Graph transformation based simulation model generation

Huang, Yilin; Verbraeck, Alexander; Seck, M

DOI
10.1057/jos.2015.21

Publication date
2016

Document Version
Accepted author manuscript

Published in
Journal of Simulation

Graph Transformation Based Simulation Model Generation

Yilin Huang, Alexander Verbraeck
Delft University of Technology, Faculty of Technology, Policy & Management
Systems Engineering & Simulation

Mamadou Seck
Old Dominion University, Batten College of Engineering & Technology
Engineering Management & Systems Engineering

The graph transformation based method presented in this paper can automatically generate simulation models assuming that the models are intended for a certain domain. The method differs from other methods in that: the data used for model generation does not contain specifications of the model structures to be generated; the generated simulation models have structures that are dynamically constructed during the model generation process. Existing data typically has quality issues and does not contain all types of information, particularly in terms of model structure, that are required for modeling. To solve the problem, transformation rules are designed to infer the required model selection and structure information from the data. The rules are specified on meta-models of the original data structure, of intermediate structures and of the simulation model. Graph patterns, pattern composites and graph pattern matching algorithms are used to define and identify potential model components. Model composite structures are represented by hypergraphs according to which simulation models are generated using model components as building blocks. The method has been applied practically in the domain of light-rail transport.

Keywords: simulation model generation; graph transformation; hypergraph; model component

1 Introduction

One of today's challenges in the field of *Modeling and Simulation* (M&S) is to model and simulate increasingly larger and more complex systems (Crosbie, 2010). It currently takes too long to develop and experiment with models not to mention the high cost and human resource involved (Fowler and Rose, 2004; Banks et al., 2010). Many examples can be found in production and manufacturing (Fowler and Rose, 2004), supply chains (Longo, 2011), air transportation (Wieland and Pritchett, 2007), health care (Mielczarek and Uzialko-Mydlikowska, 2012), to name just a few. There is a rich history of efforts to improve the effectiveness and efficiency of the modelling process, e.g., developing simulation languages and user interfaces for modelling, and developing domain specific simulators (Fowler and Rose, 2004). While they significantly reduced the time and effort in modelling, there is still considerable room for improvement (ibid.).

One opportunity is to use the available data of a system to automatically or semi-automatically generate simulation models (Fowler and Rose, 2004; Bergmann and Strassburger, 2010). Regardless of whether modeling is performed by humans or by automation, data sources include those acquired by observation and measurement, as well as documents about a system (Shannon, 1975). The former type of data can be used to determine model input distributions and to validate simulation output; the latter type can be used to define and configure simulation models. The availability of both types of data has increased along with the advances in sensor technology as well as the more popular use of computer-aided technologies such as CAD (Computer Aided Design), CAE (Computer Aided Engineerin), ERP (Enterprise Resource Planning) and MES (Manufacturing Execution Systems) systems (Glotzer et al., 2010). This has rendered automation more attractive. The increased availability of data allows for a higher degree of automation since more information becomes accessible in digital forms. At the same time, the increased amount of data often requires automation because the data can no longer be handled manually in an effective and efficient manner (ibid.).

1.1 State of the Art

In this paper, we discuss *Automated Model Generation* (AMG). The goal of our study is to develop a method that can automatically generate simulation models assuming that the models are intended for a certain domain. AMG is a relatively new research field with early works dating around 1990s[1]. Many works in AMG use circuit de-

This article is based on Huang (2013).

[1]Some literature calls it *Automated Modeling* (Amsterdam, 1993; Nayak, 1995; Xia and Smith, 1996; Granda and Montgomery, 2003).

sign schematics (Wasynczuk and Sudhoff, 1996; Eeckelaert et al., 2004; Little et al., 2010), SysML (Cao et al., 2012; Johnson et al., 2012), or bond graphs (Granda and Montgomery, 2003; Umesh Rai and Umanand, 2009; Roychoudhury et al., 2011; Tian et al., 2012; Zupančič and Sodja, 2013) to generate simulation models for physical systems such as circuits, hydraulics and mechatronics, or for biochemical processes and manufacturing (Ferney, 2000; Thomaseth, 2003; Mueller, 2007; Roman and Selisteanu, 2012). Those models are generated based on model-like descriptions that already contain the required structure information. Gelsey (1990, 1995); Levy et al. (1997) apply advanced reasoning methods to determine model structures (i.e., the relations of components) based on pre-specified component descriptions. In these works, the input for AMG, i.e., the systems specification, is prepared for the AMG, where the pre-specified parts or structures match the model parts and structures. Such approaches are not always applicable to large and complex models. Some recent works use existing data for AMG. They define general/generic models or model templates such that specific model instances can be created through parameter configuration; e.g., Brause (2004) can select differential equation models of minimum description length by parameter pruning (i.e., unnecessary parameters become zero). Harrison et al. (2004); Lucko et al. (2010) use data to configure parameters such as the amount of resources and time in workflow process models. In Wang et al. (2011), automobile general assembly plant models can be generated based on physical layout data and production data. AMS (2013) presents a pragmatic approach to AMG of automobile engine production process models using performance data.

To our knowledge, there has not been works of AMG that can *generate simulation models with flexible structures using existing data* (assuming that these simulation models are intended for a certain domain). By generating simulation models with flexible structures, we mean that models are not generated by parameter-based configuration on a pre-specified model structure but have structures that are dynamically constructed according to the existing data during AMG. Model structure variations can be achieved through parameter-based approaches (e.g., Brause, 2004; Wang et al., 2011). In such cases, possible model structures have to be pre-specified, which is not convenient and can be impractical when models are complex and/or large-scale. The ability to dynamically construct structure variations provides more flexibility. Using existing data for AMG is very different from using data that is specifically prepared for the purpose of model generation. In the latter case, the data already contains the right content and structure of information required for AMG (e.g., Ferney, 2000; Granda and Montgomery, 2003; Mueller, 2007; Roman and Selisteanu, 2012). In existing data, however, the information that can be directly used for model generation is often not readily available (COBP, 2002). The data may need to be transformed in content and structure. The transformation should eventually lead to a model structure according to which a simulation model can be generated. How to use formal methods to achieve this objective is the aim of our research. To summarize, the method presented in this paper differs from previous works in at least two aspects: (1) the data used does not contain specifications of model structures; (2) the generated models have structures that are dynamically constructed during the AMG.

1.2 Research Background

Many organizations are facing a twofold problem. They often need simulation models which take a long time to develop and incur high costs (Wieland and Pritchett, 2007; Longo, 2011). At the same time, although increasingly more data has become available which could provide useful information for M&S, much of the data is unused due to cost-effectiveness reasons (Glotzer et al., 2010). To benefit from the data for M&S, there is a need for a method that can automatically use the existing data to generate simulation models.

The running example we use in this paper is designed for HTM, an example of such organizations. The company operates a light-rail transport network in The Hague[2] (Veldhoen, 2009). Every year, a number of M&S studies is initiated by HTM[3]. New simulation models are needed to study the design and operation of new or different parts of the network. The models are different in the sense that they have different infrastructure layouts, services and timetables, etc.; yet they share similar underlying concepts. Developing the simulation models is labor intensive and time consuming: three to six months for small projects and over a year for large ones. The company possesses a large amount of data, from infrastructure design, service plans and timetables to (sensor collected) passenger counts and GPS data. Using the data for modeling could provide useful information about the system and help improve the validity of models. However, the more data modelers use, the longer it takes to develop the models. Constrained by time and cost, a large part of the data is unused.

There are huge interests from those organizations to improve such situations. Because the underlying goals of simulation projects within an organization (or a department) can be often similar (in case of HTM it is to study the infrastructure, control strategies and timetables in relation with, e.g., the quality, reliability and robustness of services), it is desirable to develop some automated routine that can generate the simulation models with different structures and to reuse some previous modeling so-

[2]HTM (www.htm.net) is a public transport operator based in The Hague, the Netherlands. Its light-rail network covers over 150 km^2 with fourteen scheduled tram lines, 140 km tracks and 540 stops.

[3]For example, the infrastructure and control at intersections (Kanacilo and Verbraeck, 2006, 2007); the design of new infrastructure and operation (Kamerling, 2007; Huang et al., 2010); the depot capacity and vehicle planning on deadhead-kilometers (non-value added trips) (Cai, 2011); the strategies in the design of infrastructure networks, service networks and timetables (van Oort, 2011).

lutions. The availability of data in those organizations makes this kind of automation possible.

Model Components The AMG method proposed in this paper uses a component-based approach[4]. The AMG process shall automatically combine model components, instantiate and configure them according to existing data so that they together constitute a simulation model. As a prerequisite for the AMG process, model components need to be developed in the first place. We choose DEVS (Zeigler et al., 2000) as the underlying modeling formalism mainly based on two criteria (de Lara et al., 2004; Vangheluwe, 2000, 2008; Wainer, 2009)[4]: (1) DEVS provides strong support for hierarchical component-based modeling, and (2) DEVS can embed and represent many other formalisms.

The model component specifications can be divided into two related parts: (I) the specification of (the behavior of) model components at the elementary compositional level, and (II) the definition of permissible (structure of) model compositions at the non-elementary compositional levels. The former provides the basic units (or building blocks) in constructing hierarchical component-based models which subsequently generate the overall model behavior. In DEVS, model behaviors at the elementary level are specified in atomic models by means of transition functions, time advance functions, etc. The behavior of a composed model is determined by the collective (interactive) behavior of its constituent components (or sub-components). Not only the behavior of the sub-components but also the alterable composite structure alter the behavior of the composed model. In DEVS, compositions are specified in coupled models. Following the concept of closure under coupling in DEVS, the resultant of coupling is also a DEVS model, which can be again used for recursive composition.

"The definition of the permissible model compositions" advocates generic definitions of model composites in order to allow for certain flexibility of composition under predefined constraints. In this sense, such definitions are not classic DEVS coupled model specifications. They are defined by patterns of DEVS coupled models through meta-models, and are accompanied by necessary functions that can create (classic) DEVS coupled models with permissible compositions (Section 4). For the case studies used in our research, we developed a model component library called LIBROS (*Library for Rail Operation Simulation*) in collaboration with HTM. LIBROS started forming in 2008 and is progressively developed and extended along with more studies for this research. Atomic models in LIBROS include rolling stocks (hereafter called vehicles), track segments, sensors, switches, signals[5] and control units. Coupled models include (composed) infrastructure components such as stations and intersections.

Steps in An AMG Process Starting from existing data to a simulation model as a final outcome, a complete AMG process shall (1) execute model transformation definitions specified on the data models and the domain meta-model, (2) generate (or instantiate) a simulation model using predefined components, and (3) perform model calibration after the initial generation of the simulation model. Accordingly, we propose to divide a complete AMG process into three steps: *Model Transformation*, *Model Instantiation* and *Model Calibration* (Figure 1). This paper focuses only on the first two steps — model transformation and model instantiation — after which a simulation model is generated with default (non-calibrated) parameter configuration[6].

Model transformation is the automatic generation of a *target model* from a *source model* according to a transformation definition (Kleppe et al., 2003; Mens and van Gorp, 2006); see Figure 2. The source (or input) model and target (or output) model conform to a source meta-model and a target meta-model respectively. A transformation engine reads the source model, executes the transformation definition, and writes the target mode, i.e., model transformations are performed on concrete models (Czarnecki and Helsen, 2006). A *transformation definition* is specified on the source and target meta-models (ibid.). It is composed of a set of *transformation rules* that together describe how a model in the source language can be transformed into a model in the target language (Kleppe et al., 2003; Mens and van Gorp, 2006). Each rule describes a small unit used to specify a transformation (Czarnecki and Helsen, 2006). Model transformation may have multiple source and target models (Mens and van Gorp, 2006). In AMG, it is possible and likely to have multiple source models which are data from different sources, but the target model is in principle a single

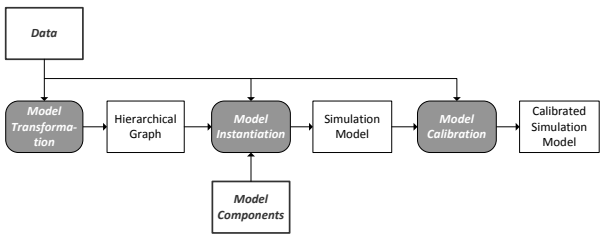


Figure 1: Proposed steps in a complete AMG process

[4]Component-based modeling is founded on a paradigm that is common to all engineering disciplines: complex systems can be obtained by assembling components (Gössler and Sifakis, 2005). It is a promising modeling concept from both systems theory perspective and M&S perspective (Huang, 2013).

[5]Sensors are used to refer to different vehicle detection and track clear detection devices used in rail operations and controls (Pachl, 2002; Theeg and Vlasenko, 2009). Switches (also called *switchpoints* or *points*) are movable track elements that are used to transfer rolling stocks from one track to another (Theeg and Vlasenko, 2009). Signals indicate if a movement may enter the section of track behind (i.e., beyond) the signaling equipment (Pachl, 2002).

[6]Readers who are interested in the final step *Model Calibration* may refer to Huang (2013, Chapter 6).

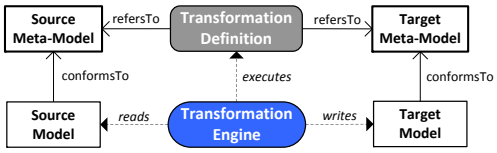
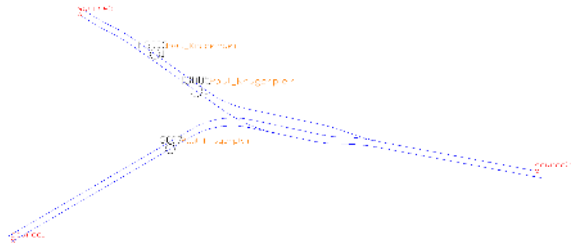


Figure 2: Basic elements in model transformation (Czarnecki and Helsen, 2006)

simulation model (for each referent of interest).

For our case studies, the (first) source meta-model is a data model, and the (last) target meta-model is a light-rail domain meta-model. We use the CAD infrastructure data provided by HTM as a basis for model structure to generate light-rail simulation models. The CAD data was originally produced as light-rail infrastructure blueprints of the *Haaglanden* region (The Hague and surroundings). Figure 3 shows a simple example which is a plot of a "Y" tram crossing. The plot visually resembles the infrastructure layout of the crossing but the CAD dataset itself only contains a list of geometric primitives that describe each CAD entity. An urban network may potentially contain hundreds of crossings that have different layouts. How to transform such flat-structured data into rich-structured simulation models was what we studied.

The main idea of the AMG method is to predefine patterns that could associate certain data structures to a set of predefined (meta-models of) simulation model components, and then use model transformation algorithms to find those patterns in the data (graph) and to replace them with the predefined components. The matching and replacement need to be performed systematically such that the whole data graph can be eventually transformed into a simulation model. Since this is often a complex process, we divided it into a few sub-steps and how to perform these steps are discussed in our paper.



The CAD Dataset:

Entity 1: line, start point, end point, color, ...;
Entity 2: line, start point, end point, color, ...;
Entity 3: arc, start point, end point, center, radius, ...;
Entity 4: circle, center, radius, color, name, ...;
...
Entity n: line, start point, end point, color, ...;

Figure 3: The CAD drawing of a "Y" tram crossing

The transformation engine (called Model Generator in Section 5) we developed reads the input data, executes the transformation rules to construct a *hierarchical graph* that is homomorphic (or isomorphic) to the simulation model to be generated. Since we use hierarchical component-based modelling and DEVS formalism, model composition can be conveniently represented by directed hierarchical graphs. In such cases, theoretical works in graph transformation are applicable to model transformation, i.e., model transformation can be treated as graph transformation. Based on the hierarchical graph as a blueprint, a simulation model can be constructed using the predefined model components as building blocks.

The rest of this paper presents the AMG process in detail. Since graph theory and graph transformation in particular are applied for the steps, the related theory is presented in Section 2. In the model transformation step explained in Section 3, transformation rules are executed on the input data in order to construct a hierarchical graph that represents the compositional structure of the simulation model to be generated. The rules are defined on the meta-models of the original data structure, of the intermediate structures, and of the domain simulation model. In the model instantiation step explained in Section 4, a simulation model is generated using the domain model components based on the hierarchical graph generated from the previous step. In Section 5, we summarize the steps and sub-steps discussed.

2 Graph Theory and Graph Transformation

In the AMG process, graphs are used to represent structures in data and model composition, to perform transformation step by step from the former to the latter, and to generate simulation models based on the latter.

2.1 Model Structure Representation with Graphs

Data can be structured, semi-structured or unstructured. Simulation models are often structured. The models discussed in this paper have hierarchical structures. This *hierarchical structure* refers to a *Compositional Containment Hierarchy* (CCH), which is a strictly nested component inclusion hierarchy. For example, component A is composed of components B and C, where C is composed of D and E while B, C, D and E do not directly belong to any other components. This structure is a *Model Composite Tree* (MCT). Using a tree to represent CCH does not convey information about the coupling relations among the model components. For this purpose, a hierarchical graph is needed.

2.1.1 Graph and Graph Pattern

A common mathematical notion of a graph is a finite set of nodes (or vertices) among which some pairs are connected by edges (or links). An enrichment of this notion is to type and/or attribute the vertices and/or edges. The following definitions are based on Ehrig et al. (2006a,b); Gallagher (2006); Fan et al. (2010).

A **directed graph** (or **digraph**) is an ordered pair $G = (V, E)$ where V is a finite set of *vertices* and $E \subseteq V \times V$ is a set of *edges* where $e = (v, v') \in E$ denotes an edge from vertex v to $v' \in V$. A similar definition is a 4-tuple $G = (V, E, s, t)$ where E is a set of edges without indications of the (source and target) vertex pairs. They are specified by the *source* and *target functions* s and t such that $s, t : E \to V$. A **path** p in a graph G is a sequence of vertices $(v_1, v_2, \cdots, v_n)$ such that $e_{i \in [1,n-1]} = (v_i, v_{i+1}) \in E \in G$.

A **typed graph** $TG = (G, T_V, T_E, \mathfrak{t}_V, \mathfrak{t}_E)$ has typed vertices and/or typed edges, where G is a graph, T_V and T_E are two finite sets of *vertex types* and *edge types*, and $\mathfrak{t}_V$ and $\mathfrak{t}_E$ are *vertex type function* and *edge type function* such that $\mathfrak{t}_V : V \to T_V$ and $\mathfrak{t}_E : E \to T_E$. A **data graph** $DG = (TG, A_V, A_E, \mathfrak{a}_V, \mathfrak{a}_E)$ is a typed attributed graph, where TG is a typed graph, A_V and A_E are two finite sets of *vertex* and *edge attributes*, and $\mathfrak{a}_V$ and $\mathfrak{a}_E$ are the *vertex attribute* and *edge attribute functions* such that $\mathfrak{a}_V : V \times A_V \to \mathbb{R}$ and $\mathfrak{a}_E : V \times A_E \to \mathbb{R}$. This means that a vertex $v \in V$ can have a number of vertex attributes $a_i \in A_V, i \in [1, n]$ each of which has an attribute value $c_i \in \mathbb{R}$, i.e., $\mathfrak{a}_V(v, a_i) = c_i$. Likewise, an edge $e \in E$ can have a number of edge attributes $a_j \in A_E, j \in [1, m]$ each of which has an attribute value $c_j \in \mathbb{R}$, i.e., $\mathfrak{a}_E(v, a_j) = c_j$. Thus, a vertex or edge can be assigned with attributes and values to carry the content such as label, rating, weight, and identifier.

Graphs can represent not only structure instances, e.g., social networks or maps, but also types or patterns of structure instances. A type of structure instances can be expressed by a *graph pattern* which is often used to describe the matching criteria of occurrence(s) of homomorphic or isomorphic subgraph(s) in a host graph.

A (typed attributed) **graph pattern** can be defined as $P = (V_p, E_p, \mathfrak{p}_p, \mathfrak{b}_p)$ where V_p and E_p are two finite sets of vertices and edges, $\mathfrak{p}_p$ is the *predicate function* defined on V_p and/or E_p as a (logical) conjunction of atomic formulas[7] over the vertex and/or edge types T_V, T_E, the vertex and/or edge attributes A_V, A_E, and the corresponding attribute values, and $\mathfrak{b}_p$ is the *bound function* such that $\mathfrak{b}_p : E_p \to \mathbb{R} \cup \infty$. Note that an edge in a graph pattern is often a path in a host graph. Intuitively, the predicate function defines the search conditions on the vertices and/or edges in a host graph, and the bound function defines the bound of a search path, e.g., the upper bound of a path length, and ∞ simply means that there is no bound.

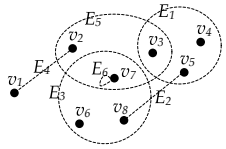


Figure 4: Hypergraph – an example with eight vertices and six edges (ibid.)

2.1.2 Hypergraph

The definitions in Section 2.1.1 do not yet support the direct expression of CCH. In order to do so, we need a mathematical notion of composition. In literature, *hyperedges* (Busatto and Hoffmann, 2001; Drewes et al., 2002; Palacz, 2004; Bruni et al., 2010) are often used for this purpose. A **hyperedge** is an edge in a *hypergraph*, a generalized graph whose edges are non empty sets of finite vertices. To our knowledge, it is first introduced by Berge (1973, 1989). Berge (1989) has the following definitions[8]. Let $V = \{v_1, v_2, \ldots, v_n\}$ be a finite set of vertices. A **hypergraph** on V is a family $H = (E_1, E_2, \ldots, E_m)$ of subsets of V such that the edges $E_i \in H, i \in [1, m]$ satisfy $E_i \neq \emptyset \wedge \cup E_i = V$. A **simple hypergraph** is a hypergraph such that no edge is contained by another, i.e., $E_i \subset E_j \Rightarrow i = j$. A **simple graph**[9] is a simple hypergraph such that each edge has cardinality 2, i.e., $|E_i| = 2$.

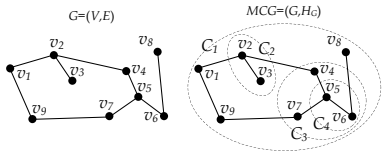
As proposed by Berge (ibid.), a hyperedge E_i may be represented by a line connecting the two vertices if $|E_i| = 2$ (similar to the case in a simple graph), by a loop if $|E_i| = 1$, and by a closed circle enclosing the vertices if $|E_i| \geq 3$ (Figure 4). One can see that a hypergraph is not per definition strictly nested. A CCH shall not have hyperedges that can freely share vertices. We can, however, impose constrains on the hyperedges to obtain CCHs.

Hyperedges and Simple Edges In representing CCHs with hypergraphs, we need to pay attention to the hyperedges that have cardinality 2. In literature, a hyperedge connecting two vertices is like an ordinary graph edge. When we use these hyperedges to represent the compositional relations of two vertices, how to distinguish them with the coupling relations of two vertices? Take the hyperedge E_2 in Figure 4 as an example. This edge connects v_5 and v_8. It can represent a compositional relation, where v_5 and v_8 are composed together to form a

[7] Atomic formulas, as opposed to composite formulas, are the simplest well-formed formulas in mathematical logic. Variables and constants are (atomic) terms; if f is an operation of degree r and $t_1, \cdots, t_r$ are terms, then $f(t_1, \cdots, t_r)$ is a term; if p is a (predicate) relation of degree r and $t_1, \cdots, t_r$ are terms, then $p(t_1, \cdots, t_r)$ is an atomic formula, which has roughly the following meaning: the ordered r-tuple of objects denoted by $t_1, \cdots, t_r$ has the property denoted by the r-ary predicate p (Manin, 2010; Ben-Ari, 2012).

[8] A more detailed concept of hypergraph is presented, e.g., in Habel (1992); Drewes et al. (1997) which include *tentacles*, *attachment vertices* and *external vertices*. We discuss them in Section 2.2.2.

[9] The (ordinary) graphs that have edges connecting two vertices only, as those defined in Section 2.1.1.



$H_G = (C_1, C_2, C_3, C_4)$; $C_1 = (v_1, v_8, v_9, C_2, C_3)$; $C_2 = (v_2, v_3)$;
$C_3 = (v_4, v_7, C_4)$; $C_4 = (v_5, v_6)$

Figure 5: Model composite graph – an example with four compositions defined on a graph with nine vertices

larger coupled component[10]. It can also represent a coupling relation, where v_5 and v_8 are simply coupled together. This ambiguity in relation definition is undesirable for model transformation. It would therefore make sense in our graph representation to distinguish hyperedges having cardinality 2 with simple edges, or at least type or label these edges, in order to represent compositional relations and coupling relations differently.

2.1.3 A Hierarchical Graph for Model Composition

Because a CCH is a strictly nested component inclusion hierarchy, we define the following constrain for the hyperedges: for each pair of hyperedges in a CCH, one edge can be a subset or superset of another edge but otherwise they must not intersect one another. Based on the definitions and discussions presented, we propose a definition of hierarchical graph that can be used to represent a CCH for model composition.

A **model composite graph** $MCG = (G, H_G)$ is an ordered pair of an ordinary digraph $G = (V, E)$ and a CCH H_G specified on G. The graph G can be typed and/or attributed as defined in Section 2.1.1. The CCH $H_G = (C_1, C_2, \ldots, C_m)$ is a family of subsets of $V \in G$ such that the hyperedges $C_i, i \in [1, m]$ satisfy: (1) $\cup C_i = V \wedge C_i \neq \emptyset$, (2) $\exists C_i = V$, and (3) $\forall C_i, C_j, j \in [1, m], i \neq j \implies C_i \cap C_j = \emptyset \vee C_i \subset C_j \vee C_i \supset C_j$.

We separate a graph G with its graph hierarchy H_G in the definition[11]. The model coupling relations are represented in G by simple edges $e \in E$. The compositional relations are represented in H_G by hyperedges C_i which are sets of composite members only. The vertices $v \in V$ and the hyperedges C_i represent the (elementary and composed) components in model composition. Figure 5 illustrates an example of *MCG* in which a hypergraph H_G with four compositions $C_1 \sim C_4$ are specified on a graph G with nine vertices $v_1 \sim v_9$. The solid lines denote simple edges and the dashed circles denote hyperedges.

Following the first condition in the definition (the same as in the hypergraph definition), each composition is a non empty set of vertices in graph G. The second condition states the existence of one component that contains all vertices in G. This is the root component that is to be instantiated as the top level model (the simulation model at the highest level in the hierarchy) in AMG. The third condition in principle states that a CCH does not allow intersected compositions. A component can be a subset or superset of another component. Otherwise, they shall not share composite members. Note that the hyperedges can be as well typed and/or attributed following definitions in Section 2.1.1.

[10] This is impermissible in case of DEVS as both vertices are also contained by other hyperedges.

[11] The definition is consistent with the hypergraph definition, e.g., in Habel (1992); Drewes et al. (1997); see Section 2.2.2. The information about attachment vertices of the hyperedges is however contained (indirectly) in the ordinary graph G; see Section 3.

2.2 Basic Concepts of Graph Transformation

2.2.1 Rules, Matches and Rule Applications

A graph transformation step involves the following basic concepts: graph transformation rules, matches and rule applications (Corradini et al., 1997; Kahl, 2002); see Figure 6 (cf. Figure 2). A **rule** (or *production*) $p : L \rightsquigarrow R$ contains at least a **left-hand side** (LHS, or *pattern graph*) and a **right-hand side** (RHS, or *replacement graph*), and some indications of how instances of the RHS are to replace the matched instances of the LHS, e.g., which vertices and edges are to be preserved, deleted and/or created. The LHS and RHS are the source and target meta-models (or subsets of them) discussed in Section 1.2.

The application of a transformation rule to an **application graph** (or *host graph*) G requires a **match** (or *graph morphism*) $m : L \to G$ for a production p. A match occurs when the vertices and edges of L can be mapped to a subgraph in G such that the defined graph structure, bound and/or the types and/or the attributes (Section 2.1.1) are preserved. This subgraph is also called an *image*.

Following Corradini et al. (1997), a **graph morphism** $f : G \to G'$ is a pair $f = \langle f_V : G_V \to G'_V, f_E : G_E \to G'_E \rangle$ of functions which preserve sources, targets, types and attributes, i.e., which satisfies $f_V \circ t^G = t^{G'} \circ f_E$, $f_V \circ s^G = s^{G'} \circ f_E$, $\mathfrak{t}^{G'}_V \circ f_V = \mathfrak{t}^G_V$, $\mathfrak{t}^{G'}_E \circ f_E = \mathfrak{t}^G_E$, $\mathfrak{a}^{G'}_V \circ f_V = \mathfrak{a}^G_V$, $\mathfrak{a}^{G'}_E \circ f_E = \mathfrak{a}^G_E$ (see Section 2.1.1). A graph morphism is an *isomorphism* if both f_V and f_E are bijections. If there exists an isomorphism from graph G to graph H, then we write $G \cong H$. $[G]$ denotes the isomorphic class of G, i.e., $[G] = \{H \mid H \cong G\}$.

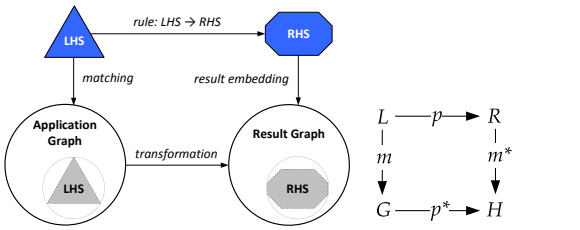


Figure 6: Graph transformation – rule-based modification of graphs (Corradini et al., 1997; Kahl, 2002; Ehrig et al., 2006b)

The **rule application** to an application graph G, produces a **result graph** (or *derived graph*) H. The transformation relation (also know as *direct derivation*) is often called *co-production* $p^* : G \rightsquigarrow H$. Roughly speaking, H is constructed as $G \backslash (L \backslash R) \cup (R \backslash L)$. The rule application can be seen as an embedding into a context which is the part of the host graph G that is not part of the match. The **result embedding** (or *co-match*) $m^* : R \to H$, maps R to its occurrence in the derived graph H.

Figure 6 (right) illustrates a schematic representation of a direct derivation from G to H, $G \overset{p,m}{\Longrightarrow} H$, resulting from an application of a production p at a match m (Corradini et al., 1997). A **graph grammar** $\mathscr{G}$ consists of a set of productions $\mathbf{P} = \{(p_i | i \in [1,n]\}$ and a *start graph* G_0. A sequence of direct derivations $G_0 \overset{p_1}{\Longrightarrow} G_1 \overset{p_2}{\Longrightarrow} \cdots \overset{p_n}{\Longrightarrow} G_n$ constitutes a *derivation* of the grammar, also denoted by $G_0 \Longrightarrow^* G_n$ (ibid.).

2.2.2 Hyperedge Replacement

Hyperedge replacement is the replacement of hyperedges of a hypergraph by hypergraphs (Habel, 1992). A directed hyperedge can be seen as a black-box or a placeholder with an ordered set of *incoming tentacles* and an ordered set of *outgoing tentacles*; Figure 7 (A) (Habel, 1992; Drewes et al., 1997). Habel (1992) gives the following definitions.

A hypergraph H with a finite set of vertices V and a finite set of hyperedges E has source and target functions $s : E \to V^*$ and $t : E \to V^*$ [12] that assign a sequence of sources $s(e)$ and a sequence of targets $t(e)$ to each $e \in E$. There is a (predefined) set of vertices occurring in the sequence $ext_H \in V^*$ which is called the set of **external vertices** of H. They (also called the *begin and end vertices*) correspond to the sequence of sources and targets of a hyperedge ($\notin E$) when H may replace this hyperedge (conditions see below). The set of all other vertices is said to be the set of **internal vertices** of H. The set of vertices occurring in the sequence $att(e) = s(e) \cdot t(e)$ is called the set of **attachment vertices** of an hyperedge e. A hyperedge $e \in E$ is called an (m,n)-edge for some $m,n \in \mathbb{N}$ if $|s(e)| = m$ and $|t(e)| = n$. The pair (m,n) is the *type* of e, denoted by $type(e)$. Figure 7 (B) illustrates a hyperedge where $type(e) = (m,n)$ whose attachment vertices (i.e., source and target vertices) are denoted by dots ($\bullet$).

Given hypergraphs H and R, a hyperedge $e \in E \in H$ may be replaced by R when e and R "fit together"; this means that when e and R are of the same type — if e is an (m,n)-edge then R is an (m,n)-hypergraph — and whenever the i-th and the j-th external vertices of R are the same then the i-th and the j-th attachment vertices of e are the same (i.e., distinct tentacles of a hyperedge may be attached to the same vertex, ibid.). The hyperedge replacement can be done by removing the hyperedge e, adding the hypergraph R except its external vertices, and handing over each tentacle of each hyperedge (in the replacing hypergraph R) which is attached to a begin or end vertex to the corresponding source or target vertex of the replaced hyperedge (ibid.); see Figure 8.

A formal definition of hyperedge replacement can be found in Habel (ibid.). Andersson (2006) provides a simpler definition as following. Let H, R be hypergraphs, $e \in E \in H$, $type(e) = type(R)$. The replacement of e in H by R yields the hypergraph $H[e/R]$ which is obtained with three steps: (1) build $H \backslash e$ by removing e from H; (2) take the disjoint union of $H \backslash e$ and R, i.e., $H \backslash e \uplus R$; (3) $\forall i \in [1, type(e)]$, identify the i-th external vertex of R with the i-th attachment vertex of e. This means that when adding R to $H \backslash e$, the sequence of external vertices of R is fused with the sequence of attachment vertices of e in the right order (Drewes et al., 1997).

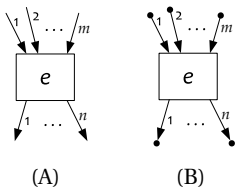


Figure 7: Hyperedge of type (m,n). (A) A hyperedge with m incoming tentacles and n outgoing tentacles. (B) A hyperedge with m sources and n targets (Habel, 1992)

[12] For a set A, A^* denotes the set of all strings over A including the empty string.

3 Model Transformation

In this section, we explain and exemplify how to define transformation rules that are executed on the input data (i.e., a start graph G_0; Section 2.2.1) to construct a hierarchical graph (i.e., a model composite graph *MCG*; Section 2.1.3) that represents the compositional structure of the simulation model to be generated.

Typically, existing data has quality issues and does not contain all types of information, particularly in terms of model structures, that are required for AMG. An $MCG = (G, H_G)$ is structured where G shall be a graph whose edges represent (directed) model coupling relations. The hyperedges in H_G shall represent the structure of each model component. They are the results of *Graph Pattern*

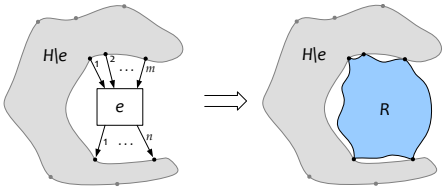


Figure 8: Hyperedge replacement $H \Rightarrow H[e/R]$ (ibid.)

Matching (GPM) where different types of data structure combinations (each of which has a corresponding model component) are defined as graph patterns. In designing the transformation rules, the information gap should be identified and measures are defined accordingly (often step by step) to close this gap. The rules basically include the measures that are automatable.

3.1 Start Graph

Figure 9 shows a plot of the CAD infrastructure data we use as a basis for model structures. The data contains a list of geometric primitives (each of which has a shape with geometrical descriptions and possibly other descriptions such as colour and label). Although the data plot appears to be a network visually, the dataset itself is unstructured, i.e., it does not contain relations among the entities. As such, it is hardly a graph as a mathematical structure. When considered as a graph, it has vertices which are the CAD entities but has no edges.

With such a start graph G_0, a set of productions (or transformation rules) **P** is defined to derive an *MCG*. (The final production from the *MCG* to a simulation model is discussed in Section 4.)

3.1.1 Solving Data Quality Issues

A number of data quality criteria could help detect data quality issues for AMG (Huang, 2013, Section 3.3). This subsection briefly discusses some measures.

Syntactic accuracy relates to lawfulness rather than correctness of data values (Wand and Wang, 1996). It often can be automatically checked by comparison functions (Batini and Scannapieco, 2006). *Syntactic consistency* issues are particularly relevant when data has multiple sources (Shanks and Corbitt, 1999). They can be typically solved through type and format conversion.

To measure *semantic accuracy* of a data value v, (i) the corresponding true value v' has to be known, or (ii) it should be possible, considering additional knowledge, to deduce whether v is or is not v' (Batini and Scannapieco, 2006). The first option is a non-option in a computational sense: if the "true value" is or can be known digitally, that value should be used instead of v. Hence, semantic accuracy is only computationally measurable and solvable with sufficient knowledge to reason the deduction.

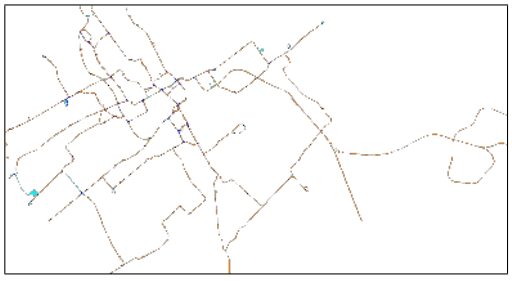


Figure 9: Light-rail infrastructure CAD data of the *Haaglanden* region

Data completeness issues can be semantic or pragmatic. When data is *truly* incomplete[13], the only way to complete the data is to acquire the missing part. Improving semantic completeness could potentially increase pragmatic completeness. Semantic incompleteness does not necessarily signify pragmatic incompleteness.

Mapping consistency issues typically occur among data from different sources. Sometimes consistency is broken because of erroneous schema changes (Velegrakis et al., 2004). When key values intended to map to the same external instance are inconsistent, a mapping table can clarify the relations among these keys.

Presentation suitability is the degree to which the data format, unit, type-sufficiency are appropriate for the purpose of data use. (Type-sufficiency is the degree to which the data includes all types of information useful for the purpose of data use.) Existing data often does not contain the right content and structure of information required for model generation, i.e., the parts and relations in the simulation model is richer than those represented in the data. We do not deem this issue as being semantic incomplete, because the missing information can be deduced from the existing data with sufficient domain knowledge. The data is not (truly) incomplete but the information directly contained is not of the right type. When the domain knowledge and reasoning for deduction can be formalized, we are able to obtain the right type of information automatically from the data.

Quality Issues in the Infrastructure Data The data we use for case studies contains a number of quality issues. Since the CAD entities were drew manually as light-rail infrastructure blueprints, many data quality issues concern semantic accuracy. For example, (1) two entities may appear to be connected visually but are indeed unconnected which can be inspected only when zoomed in with a sufficiently large scale factor; (2) there exist small "invisible" entities, e.g., very short lines and arcs, that shall not be considered as a part in model transformation; (3) although the entities representing rail tracks have start and end points, they do not correspond to traffic directions, which are important for the model.

To solve these issues, we used a number of measures. For issue (1), a configurable parameter for *snap tolerance* is used to evaluate the connectedness of the CAD properties. For issue (2), we "traverse" the infrastructure entities for connectedness and cleaned the entities that are not needed. As for issue (3), the direction of traffic is not an intrinsic track property that can be inferred from the track geometry alone. But if the origin of a traffic flow can be indicated in some way, and given that a rail track has a unique direction of traffic, it is possible to infer the direc-

[13] This means that (i) the data values or records are unknown but do exist, (ii) they are not contained by other accessible data sources, and (iii) they are not deducible from known data values or records.

tion of traffic of the successively connected tracks. Urban light-rail services largely operate on rails with dedicated directions (Pachl, 2002; Vuchic, 2005); this is the case in the *Haaglanden* region. The origins of traffic flows, called sources, are typically at the boundary of a modelled area or at defined locations such as terminals. Since there is a very low number of sources, we manually added this information into the original infrastructure CAD data using labelled circle entities.

3.1.2 Defining Information Types and Dependencies

Major data quality issues that shall be solved by the transformation steps concern presentation suitability. To solve the information gap, we need to first identify what information is missing and how to obtain it. We used an intuitive approach. The types of information that are available are enumerated against those that are required. We arrange the information types (Info. Types) by their dependencies and try to fill in the gaps by adding the information that can be inferred from those that are (or can be) known. Expert opinions and literature are consulted in order to complete and verify the information dependencies and inference logic. The identified information types are arranged into groups, each of which represents an intermediate transformation step. The dependencies are used as a basis for the design of the stepwise transformation. Each step has a source graph and a target graph. Each graph shall contain the information types that are specified respectively. For the light-rail case, we identified twelve information types with four groups (Figure 10). To obtain a LIBROS model, three transformation steps (i.e., three productions) are needed according to the dependencies: $G_0 \overset{p_1}{\Longrightarrow} G_1 \overset{p_2}{\Longrightarrow} G_2 \overset{p_3}{\Longrightarrow} G_3$.

3.2 Transformation Step 1: Generation of Digraph

In the first step, a production p_1 is applied on the start graph G_0 resulting a digraph G_1. G_0 is a non-graph composed of a list of geometric primitives (Section 3.1). G_1 shall contain Info. Types 1∼5[14] as shown in Figure 10.

The construction of G_1 basically relies on geometrical inference. We pre-process G_0 to partition its entities into three non overlapping *sets* of sources, tracks and stops. In the light-rail domain, the rail infrastructure has intrinsic characteristics that allow for a limited number of entity-to-entity compositions with regard to vertex types and degrees. Figure 11 (left) illustrates the schema of such compositions. Note that the white-headed arrows represent track entities that are vertices but not edges, and their directions indicate permissible directions of the

[14]This step shall construct a digraph G_1 whose vertices are of two types: sources or tracks (Info. Types 1 and 2). The geometrical compositions of the tracks are indicants of locations and types of points (Info. Type 4). Additionally, since a stop model contains a sequence of connected tracks with a certain total length and this information is already available at this step, hyperedges are defined on G_1 to represent the stops and their constituent tracks (Info. Types 3 and 5).

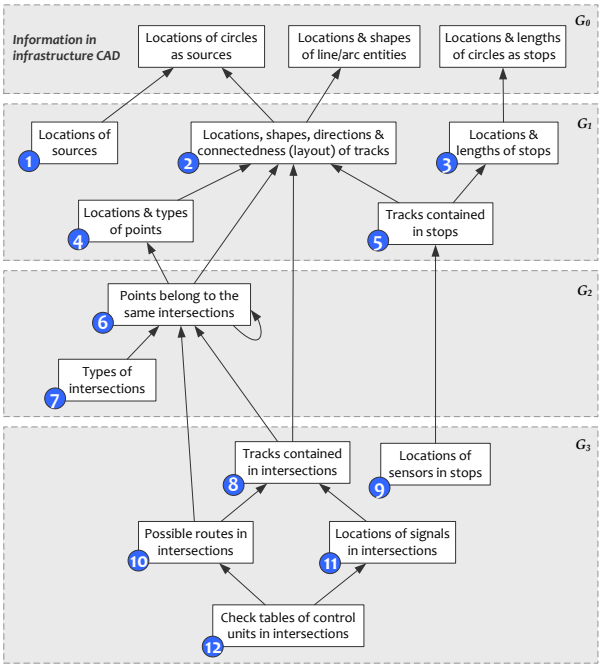


Figure 10: Composite information types and dependencies for the AMG of LIBROS models

traffic which are not contained in the data as such but need to be inferred during model transformation.

The transformation in this step is completed with one "graph traversal" during which the graph edges are created based on the entities' geometric connectedness. We apply *Depth-First Search* (DFS) for the traversal since DFS can be used to classify the edges (Cormen et al., 2001) which suits our situation of exploring G_0. In the traversal, the search root of a new (depth-first) tree in the depth-first forest (ibid.) is always a source vertex.

3.2.1 Search for Connectedness

The search of entities' geometric connectedness (to a reference point) takes place (along the traversal) with a given snap tolerance; both the start and end points of a track entity (which is a line or arc entity) are checked for connectedness. An ordered pair of connected entities[15] becomes an edge in digraph G_1. A discovered (or visited) edge is recorded in a map-like data structure, let us call it *track map* **T**, in which the leading vertex is an index (or a key) and the following (track) vertex is an indexed value[16].

Starting from a source vertex, the connected track vertices are searched in the track set. A valid source shall have exact one track connected to it. A track entity shall have its start and end points correspond to the permissible direction of the traffic; if not, the these two points must be swapped. We call this operation *regulating track*

[15]It would be a (source, track) pair or a (track, track) pair.
[16]An index can be associated with more than one value.

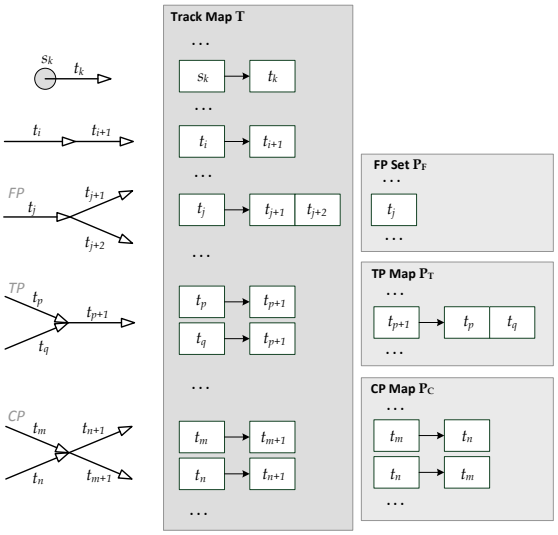
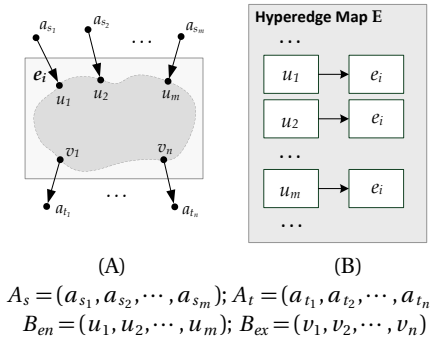


Figure 11: Representation of track composition information in set and maps



$A_s = (a_{s_1}, a_{s_2}, \cdots, a_{s_m})$; $A_t = (a_{t_1}, a_{t_2}, \cdots, a_{t_n})$
$B_{en} = (u_1, u_2, \cdots, u_m)$; $B_{ex} = (v_1, v_2, \cdots, v_n)$

Figure 12: An (m,n)-edge e_i representing a composite with m entry vertices and n exit vertices. It has m entries in the hyperedge map

direction. A track with a regulated direction is called a *regulated track*. Starting from a regulated track vertex (with its end point as the reference point), the connected track vertices are again searched in the track set. A track may have non, one, two or three connected tracks, which may correspond to the situation of being a sink, a normal track, a stop, a facing point (FP), a trailing point (TP), or a crossing point (CP) depending on the geometry[17].

The DFS proceeds each time when there is a connected track and terminates when all sources and connected tracks are visited. After the traversal, a track map **T** holds the information about the (source) track relations (Info. Types 1 and 2); together with a FP set $\mathbf{P_F}$, a TP map $\mathbf{P_T}$ and a CP map $\mathbf{P_C}$, they hold the information about the points (Info. Type 4). Figure 11 illustrates the representation of the information in set and maps, with which G_1 is described. A hyperedge map **E** (Section 3.2.2) shall hold information about all (potential) components; at this stage it only holds information about the stops which are composites (Info. Types 3 and 5). As the outcome of transformation step 1, G_1 can be expressed such $G_1 = (\mathbf{T}, \mathbf{P_F}, \mathbf{P_T}, \mathbf{P_C}, \mathbf{E})$.

3.2.2 Hyperedge Representation of Composition

Let e be an (m,n)-edge with a set of source attachment vertices $A_s = (a_{s_1}, a_{s_2}, \cdots, a_{s_m})$ and a set of target attachment vertices $A_t = (a_{t_1}, a_{t_2}, \cdots, a_{t_n})$. Suppose edge e can be replaced with a given (m,n)-hypergraph, say H_e (Section 2.2.2). To generate H_e, we need to know the composition of H_e without saying that the external vertices of H_e shall match the attachment vertices of e. This composition is described as a subgraph in G_1. We still need to define the "boundary" of this subgraph.

Can we use the attachment vertices of e as the boundary? Certainly we can. But since the attachment vertices may connect with other vertices that are not a part of e, we would need extra operations to check this through. Additionally, the attachment vertices may take part in other hyperedges. It is conceptually inexplicable to use members of a hyperedge as boundaries of another hyperedge. Hence, we use an equally easy but more straightforward way to define the boundary: using the "outermost vertices in e". They are the vertices that shall belong to H_e and have edges with vertices that shall be external vertices of H_e. We call these vertices *boundary vertices* B of hyperedge e, where $B \in V \in e$. The boundary vertices that are connected by source attachment vertices are *entry vertices* B_{en}; those that are connected to target attachment vertices are *exit vertices* B_{ex}, i.e., $B_{en} \cup B_{ex} = B \in V, B_{en} \cap B_{ex} = \emptyset$. Figure 12 (A) illustrates an (m,n)-edge e_i with corresponding attachment and boundary vertices. Although the figure only shows one-to-one attachment-boundary vertex connections for simplicity, this is not a vital condition. An attachment vertex can connect with more than one boundary vertices and vice versa. The two sets B_{en} and B_{ex} (specifying the boundary of e_i) are needed for generating the replacement graph H_e. We therefore extend the hyperedge definition in the *MCG* (Section 2.1.3) with these two sets, while the incoming and outgoing tentacles connecting A_s with B_{en} and B_{ex} with A_t respectively are already specified in the track map **T**.

Hyperedge Map The hyperedges created during the transformation steps are recorded in a hyperedge map **E**. The edges are indexed by their entry vertices such that for *each* entry vertex $u_i \in B_{en}$ in a hyperedge e, there exists one map-entry (u_i, e) in **E**. This means that the entry vertices are not used as joint but independent indexes, and each (m,n)-edge has m entries in **E**, as illustrated in Figure 12 (B). Note that a hyperedge is defined by its entry and exit vertex sets together with the composite in-

[17] Details can be found in Huang (2013, Section 5.2.2).

formation about the internal vertices contained in G_1 by the set and maps. The hyperedge map $\mathbf{E}$ serves to fasten the search of hyperedges during pattern composite (Section 3.3.2) and model generation (Section 4). When the search is in the order of the traversal (or in the direction of the traffic in our case), it is sufficient to index hyperedges with entry vertices. In case modelers need to search in a reversed order, hyperedges can also be indexed by their exit vertices.

3.3 Model Composite Graph

An *MCG* is composed of an ordinary digraph G and a CCH hypergraph H_G specified on G. After transformation step 1, we obtain the ordinary digraph, which is $\mathbf{T} \in G_1$, and a part of the CCH hypergraph, the latter being a number of stop hyperedges contained in $\mathbf{E} \in G_1$. In transformation step 2 (Section 3.4), we need to create more hyperedges that represent other model components to complete H_G. For the light-rail case, it contains Info. Types 6 and 7. Since the other hyperedges have compositions that are not as simple as a stop, we defined graph patterns to search for the occurrences of these composites in G_1. We observed recurrence of some small graph patterns in larger ones. In order to simplify and reuse the graph patterns and the corresponding search algorithms, we used graph pattern composition (recursive definition and incremental search) in the transformation of the CCH hypergraph.

3.3.1 Graph Patterns and Pattern Composites

The rail infrastructure has a number of characteristics in its geometric composition. Three basic composites are the FPs, TPs, and CPs. They are used as the basic units to define the graph patterns for the GPM in transformation step 2. There are many rail infrastructure layouts. We shall describe a variety of layouts with a limited number of pattern definitions. Figure 13 shows the rail composites we choose to define[18]. The gray boxes denote different types of hyperedges that shall be specified for the composites. Each (m,n)-edge will be transformed into a (coupled) model component (Section 4). Note the not

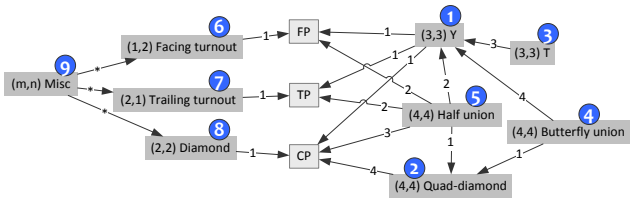


Figure 13: Rail composites and their dependencies

each type of composite needs a pattern. Types 6∼8, e.g., are simply one-to-one mappings from the points, and Type 9 is so general that it is hardly a pattern. The information about FP, TP and CP is contained in $\mathbf{P_F}, \mathbf{P_T}$ and $\mathbf{P_C}$ after transformation step 1. The arrows denote composite (information) dependencies or aggregation relations between the composites. In defining the composites, some composites are reused in larger ones; e.g., the "Y" composite (Type 1) contains one FP, one TP and one CP; the "T" composite (Type 3) contains three "Y" composites, while the butterfly union composite (Type 4) contains four "Y" composites.

3.3.2 Representing Graph Patterns and Pattern Composites

The main task in transformation step 2 is to search certain composites in G_1. To do so, we define them in graph patterns. The "Y" composite, e.g., is a (3,3)-edge representing a common rail arrangement[19]. Its schema is illustrated in Figure 15. (The gray lines are the tentacles which are not a part of the composite.) A solid-lined arrow denotes a track vertex. A dot-lined arrow denotes a unique independent path that connects the two corresponding terminal vertices. In an **independent path** (also called *path graph* or *liner graph*) the internal vertices of the path do not incident edges other than the edges in the path. In other words, if the path has internal nodes, the in-degree (d^+) and out-degree (d^-) of each internal node are both 1. The "Y" composite has two unique and independent paths p_1 (connecting the FP to the CP) and p_2 (connecting the CP to the TP).

Ordered Graph Isomorphism Note that the geometric arrangement of the composites in our study have to be preserved in GPM. This means that a matched image of a pattern can have, e.g., rotations, but there shall not be mirroring (or flipping) in the image or of the image as a whole. For example, if there exists an "image" that has an "equivalent" of p_1 (Figure 15) which connects fp_{ex2} to cp_{en2} instead of cp_{en1}, then this is *not* a match, which would be considered as a match in a general graph where geometric information is typically not represented (Jiang and Bunke, 1996). The composites we use belong to a special class of graphs. In graph theory, they are called *ordered graphs* (Jiang and Bunke, 1996, 1999). In an **ordered graph**, the edges incident to a vertex are uniquely ordered; in a plane graph, e.g., the ordering can be clockwise or counterclockwise (Jiang and Bunke, 1999). An *ordered graph isomorphism* is generally constrained, in which the ordering property is preserved (ibid.).

Following Jiang and Bunke (ibid.), an *ordered graph* is a triple $G = (V, E, L)$ where (V, E) defines a graph; for

[18] They are not complete to define all rail infrastructure layouts but are deemed sufficient to define the infrastructure of HTM light-rail services for our studies.

[19] This arrangement is sometimes called *double junction*. It is where a double track railway splits into two double track lines. A *double track railway* runs one track in each direction, compared to a *single track railway* where trains in both directions share the same track. In our case, the double track is *right hand running*.

each vertex $v \in V$, the edges $(v,v_1),(v,v_2),\cdots,(v,v_k)$ incident to v have a unique order represented by a cyclic list $L(v)$. Two ordered graphs $G = (V,E,L)$ and $G' = (V',E',L')$ are *isomorphic* if there exists an isomorphism f between the two graphs (V,E) and (V',E') such that the order is preserved; that is, if for any vertex $v \in V$, we have $L(v) = \langle (v,v_1),(v,v_2),\cdots,(v,v_k)\rangle$, then $L'(f(v)) = \langle (f(v),f(v_1)),(f(v),f(v_2)),\cdots,(f(v),f(v_k))\rangle$ holds (1999).

Jiang and Bunke (ibid.) propose an algorithm that can optimally solve the ordered graph isomorphism problem[20] in quadratic time, i.e., $O(n^2)$. In our case, the rail composites have more constraints than general ordered graphs: the rail composites are planar and sparse (with specific geometry), and they have bounded vertex degrees and bounded path distances. We therefore choose to take advantage of these properties by designing an algorithm that walks the candidate subgraphs (which therefore only takes linear time) to solve the isomorphism problem (Section 3.3.3).

Ordering Track Compositions of Points In preparation for ordered GPM, we order the composites by first ordering the track compositions of points (Figure 14). The tracks around the point center are simply ordered by their directions, viz., entry (*en*) or exit (*ex*). When there are two entry or exit tracks, we take the non-reflex angle θ ($0 < \theta < \pi$) as reference, and number the tracks (the two angle sides) counterclockwise[21] as 1 and 2. Note that the geometry of a point naturally imposes a unique orderins of the tracks surrounding it.

Representing (Standard) Composites with Ordering Using the above orderings, we can define (standard) rail composites which logically have unique orderings because the composites are planar, and the tracks surrounding the points are ordered, and the paths between them (if any) are independent. The graph pattern defined for the "Y" composite, e.g., contains two independent paths connecting three points[22] in a specific order (Figure 15). The (3,3) type implies the number of entry and exit vertices whose mapping relations (in case of a match) are specified. The (constituent) composites and paths specify the matching conditions. In addition, a bound b is specified to constrain the search scope. In our case, this bound is the max search distance[23] of each path.

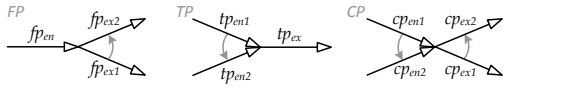
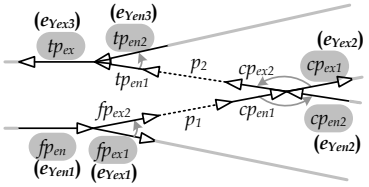


Figure 14: Ordering of track compositions of points



"Y" composite (3,3) e_Y		
Composites		fp, cp, tp
Paths	p_1	fp_{ex2} to cp_{en1}
	p_2	cp_{ex2} to tp_{en1}
Bound		$b \in \mathbb{N}$
Entry	1. e_{Yen1}	fp_{en}
	2. e_{Yen2}	cp_{en2}
	3. e_{Yen3}	tp_{en2}
Exit	1. e_{Yex1}	fp_{ex1}
	2. e_{Yex2}	cp_{ex1}
	3. e_{Yex3}	tp_{ex}

Figure 15: "Y" composite defined as a graph pattern ϱ_Y

A GPM algorithm only needs to search for the existence of the points and paths in $\mathbf{P_F}, \mathbf{P_T}, \mathbf{P_C}$ and $\mathbf{T}$ respectively according to the pattern definition. When there is a match, a hyperedge is created to represent the occurrence of the composite by specifying the boundary vertices accordingly. A created hyperedge is recorded in the hyperedge map $\mathbf{E}$ indexed by its entry vertices.

Planar composites (recursively) defined by ordered planar composites and independent paths are also uniquely ordered. The graph pattern defined for the "T" composite[24], e.g., contains three "Y" composites connected by six paths in a specific order. Its pattern definition has the same form as the "Y" composite example, but the definition is recursive.

Representing Non-Standard Composites with Ordering The "Y" composite is an examples of *regular* or *standard* rail composites which have "fixed" geometric arrangements that are known a priori. But not all parts of rail infrastructure have standard arrangements. Representing (automatically) non-standard rail infrastructure parts, we need definitions that can have "unfixed" (constituent) composites with ordering. For this purpose, we use a misc (miscellaneous) composite which is an (m,n) type

[20] Jiang and Bunke (1999)'s algorithm uses encoding of Eulerian circuits of ordered graphs starting with an edge in an graph. Since the code depends on the choice of the starting edge, $2n$ codes can be generated for some ordered graph of n edges, which can uniquely represent the graph. Ordered graph isomorphism is determined by comparing the codes of two graphs (whether isomorphism exists) and checking the order-preserving property of the two (ibid.).

[21] Since there is always one track on the right side and the other on the left when we rotate the angle θ pointing downwards, we aliased the Nr. 1 track as "right" (R) and the Nr. 2 track as "left" (L).

[22] Points with ordering means that they are represented by ordered tracks, i.e., $fp = (fp_{en}, fp_{ex1}, fp_{ex2})$, $tp = (tp_{en1}, tp_{en2}, tp_{ex})$, $cp = (cp_{en1}, cp_{en2}, cp_{ex1}, cp_{ex2})$.

[23] This distance means the geometric distance of a path which is the sum of the lengths of all track vertices in the path. We use "distance" to distinguish with the commonly used "length of a path" in graph theory which refers to the number of edges in a path.

[24] This and more pattern definitions can be found in Huang (2013, Appx. C.2).

container what does not have any specific (constituent) composite. We designed an algorithm (Section 3.4.2) that can cluster points that are close enough to one another (given a predefined distance), and each cluster of points is put into a misc composite.

3.3.3 An Algorithm for Composite Isomorphism

A **composite pattern** with boundary can be defined as $\varrho = (t, C, P, b, f_{en}, f_{ex})$ where

- $t = (m, n),\ m, n \in \mathbb{N}$ is the type of ϱ;
- $C = (c_1, c_2, \cdots, c_w),\ w \in \mathbb{N}$, is a set of pairwise distinct (constituent) composites whose entry and exit vertices are ordered;
- $P = (p_1, p_2, \cdots, p_q),\ q \in \mathbb{N}$, is a set of paths, in which $p_i = (ex_i, en_i),\ i \in [1, q]$ is an independent path that connects an exit vertex of a composite c_{a,ex_x} ($ex_i = c_{a,ex_x}$) to an entry vertex of another composite c_{b,en_y} ($en_i = c_{b,en_y}$) where $a \neq b \in [1, w]$;
- $\boldsymbol{b} \in \mathbb{N}$ is a fixed upper bound of the distance of a path p_i denoted by $d(p_i)$, i.e., $d(p_i) \in [0, \boldsymbol{b}]$.
- f_{en} and f_{ex} are the entry and exit vertex mapping functions respectively that maps entry and exit vertices of constituent composites in C to the entry and exit vertices of the composite.

In a valid composite pattern definition, the size of the entry and exit vertex mapping (by f_{en} and f_{ex}) must match the pattern type (m, n). In addition, all entry and exit vertices of the constituent composites in a pattern must be either connected by a path or mapped to the boundary vertices of the pattern. This means, a pattern ϱ necessarily has the two properties[25] $\sum_{a=1}^{w} m(c_a) = q + m$ and $\sum_{a=1}^{w} n(c_a) = q + n$.

Path Sorting The paths $P \in \varrho$ shall be *sorted* such that a composite c_a that contains $ex_j \in p_j \in P$ ($j \in [2, q]$) already "occurred" in a previous path $p \in (p_1, p_2, \cdots, p_{j-1})$ in the sense that c_a contains a terminal vertex of p. This means that the start vertex ex_j of a path p_j (except for the first path p_1) is known at the moment of searching for path p_j. This condition is not necessary for the validity of the composite pattern definition but it affects significantly the performance of the search algorithm. The path sorting problem can be reduced to the well known *topological sort* (Cormen et al., 2001). A topological sort is possible if and only if the graph has no directed cycles (ibid.), which is the case of all rail composites we defined. The paths in the "Y" composite definition, e.g., are topologically sorted.

Given a composite pattern ϱ whose path definition P is topologically sorted, a *Composite Pattern Matching* (CPM) algorithm (Alg. 1 Appx. A) is defined to search for all isomorphs of ϱ in a host graph $G = (\mathbf{T}, \mathbf{P_F}, \mathbf{P_T}, \mathbf{P_C}, \mathbf{E})$. In principle, it does not traverse G per se, but searches the occurrences of the (constituent) composites $c'_1, c'_2, \cdots, c'_w$ whose candidates are contained in composite maps $(\mathbf{P_F}, \mathbf{P_T}, \mathbf{P_C}, \mathbf{E})$ through walking the independent paths (contained in $\mathbf{T}$) that connect them.

Since the host graph and the (planar) composite patterns are ordered, and the paths in the composite patterns are topologically sorted, the CPM algorithm in the worst case takes liner time $O(n)$ to the (edge) size of the host graph, as it will walk the entire host graph, by which the performance is comparable to that of a graph traversal. The average performance is often substantially better than the worst case because given the rail infrastructure graph and composite patterns we use, the matches and partial matches of patterns rarely spread densely over the whole host graph.

3.4 Transformation Step 2: Generation of Model Composite Graph

Transformation step 1 produces a directed graph $G_1 = (\mathbf{T}, \mathbf{P_F}, \mathbf{P_T}, \mathbf{P_C}, \mathbf{E})$ with bounded vertex degrees. For this step, we need to design a production p_2 that applies the CPM algorithm to construct G_2 which is an *MCG*. As a preparation for applying CPM, the track composites for points (in $\mathbf{P_F}$, $\mathbf{P_T}$, $\mathbf{P_C}$) are first ordered. The composite patterns are defined, and the paths in each pattern are topologically sorted.

3.4.1 Rule Application Control

The transformation in this step is composed of partially ordered sub-steps that are designed on the basis of the model composite dependencies (Section 3.3.1)[26]. The sub-steps are ordered such that the required composites in a step are prepared by one or more previous steps. During the CPM, a match is followed by a rewriting where an (m, n)-edge representing the match is created and the corresponding constituent composites are removed from the composite maps.

Incremental Pattern Matching Transforming matches into hyperedges allows us to simplify pattern definitions, match routines and to aggregate match results. Our approach starts with the matching of smaller graph patterns during which the matches are transformed into intermediate structures (i.e., the hyperedges) based on which the matching of larger (higher-level) graph patterns are performed. The pattern definition takes into consideration and takes advantage that the CCH of the *MCG* is strictly nested so that a bottom-up approach of CPM can be performed on the graph. In addition, since the infrastructure graph is sparse and has bounded vertex degrees (and types and orders) surrounding which the patterns occur, the search space of the CPM is reduced to the locations of points, the independent paths that connecting them,

[25] For simplicity, we use $m(c)$ and $n(c)$ to denote the number of entry and exit vertices of c respectively, where c can be a composite pattern or a match of a composite pattern.

[26] Details can be found in Huang (2013, Section 5.2.4).

and their composites. For example, after the CPM algorithm is applied to detecte "Y" composites, each matched "Y" composite is transformed into a (3,3)-edge of type e_Y and recorded in **E**. Afterwards, for detecting "T" composites (each composed of three "Y" composites), the hyperedges of e_Y are the candidates. We only need to search for the paths between the boundary vertices of the "Y" composites without stepping into the "Y" composites.

We use a hyperedge e to represent a match denoted by $e = (C', P', B_{en}, B_{ex})$ where all images of the constituent composites, paths, boundary (entry and exit)vertices are stored according to the pattern definition ϱ. The constituent composites are removed from their original container, i.e., the composite maps. The newly constructed e is indexed in **E** by its entry vertices B_{en}. Note that since the matched constituent composites in a previous sub-step are removed from their original container, they do not appear in a later sub-step.

Automorphism in Composite Patterns An automorphism of a graph G is a graph isomorphism from G to itself (Weisstein, 2009). An ordered automorphism is an automorphism in which the ordering property is preserved. For example, each point (see Figure 14) has automorphism but not ordered automorphism. There are (ordered) automorphisms in the composite patterns we defined, e.g., the "T" composite ϱ_T.

How does this matter to CPM? It depends. When a composite pattern ϱ has automorphism, its matches or images also have automorphisms. How the image is ordered matters if this image is used further as a constituent composite in a higher-level composite. Consider a directed ordered square graph s with its four vertices numbered as $1 \sim 4$ (Figure 16). s has automorphism, and each element of the automorphism group $Aut(s) = \{(1,2,3,4), (2,3,4,1), (3,4,1,2), (4,1,2,3)\}$ is isomorphic to another (the vertices has *rotations*). When s is embedded into a larger graph, say s', there are four different ways to position s (as an ordered graph), i.e., with edges $(1,2)$, $(2,3)$, $(3,4)$ or $(4,1)$. The question is: do we want to deem them as the "same" in ordered graph isomorphism?

It depends on the applications. The CPM in our application shall deem them as same. In this case, automorphism poses a problem when *a composite pattern ϱ contains a (constituent) composite c that has automorphism.* Note that it does not matter whether ϱ itself is automorphic so long that it is not contained in another pattern. How to deal with this problem? There can be different solutions. The one we use is to reposition the matches of the automorphic composite, and reapply the CPM algorithm. This simply means rotate the match (as in the square graph example, Figure 16) of the hyperedges, which is to rotate the entry and exit vertices respectively. This is a very cheap solution in computation[27].

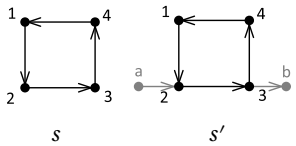


Figure 16: Automorphism of a directed ordered square graph s and its embedding s'

Composition and Decomposition in Transformation Pattern composition (and incremental pattern matching) allows modelers to make stepwise bottom-up definition and application of transformation rules. A good companion of pattern composition is decomposition, which allows composition definitions to be more flexible. Some composites can be useful as candidates for being a part in some higher-level composites. But once they are disqualified as a match in the higher-level composites, we may not need them as composites by themselves. These composites can be decomposed into lower-level composites, which in turn may be used to compose other composites.

For example, in transformation step 2, some matches of "Y" composites, even when not contained by a larger composite (e.g., a "T" composite), do not constitute stand alone crossings (or intersections) because some points in these "Y" composites are closely surrounded by other points which shall together form misc composites. These "Y" composites are therefore decomposed[28] in order to make the constituent points available for further composition of misc composites.

3.4.2 An Algorithm for Misc Composites

The *Miscellaneous (Misc) Composite Finding* (MCF) algorithm performs the following: given a set of point composites E_P (i.e., hyperedges of type e_F, e_{TR}, e_C which represent facing, trailing and crossing points respectively[29]), return a set of misc composites E_M such that the elements in E_P, connected by independent paths each of which within a bounded distance $\boldsymbol{b}$, are merged into the same element in E_M where each element in e_M shall be ordered. The main tasks here are (1) path search, (2) merging and (3) ordering. The search of independent paths with bounded distance is straightforward. It is discussed in the CPM algorithm (Appx. A). This time, the terminal vertices must be boundary vertices of the point composites. We use DFS again (as in transformation step 1) to walk the paths (in subgraphs). Merging and ordering are performed along the walk when a qualified path is found. The merge concept is similar to agglomerative clustering with single linkage (Manning et al., 2008), where the individual point composites are merged by

[27] Details of this solution and other possible solutions can be found in Huang (2013, Section 5.2.4).

[28] The rewriting of the decomposition is rather simple which is a reverse of creating e_Y: the hyperedge is removed from **E** and the points are put back into $\mathbf{P_F}$, $\mathbf{P_T}$, $\mathbf{P_C}$ respectively.

[29] We hereinafter use e_P instead of (e_F, e_{TR} and e_C) for simplicity.

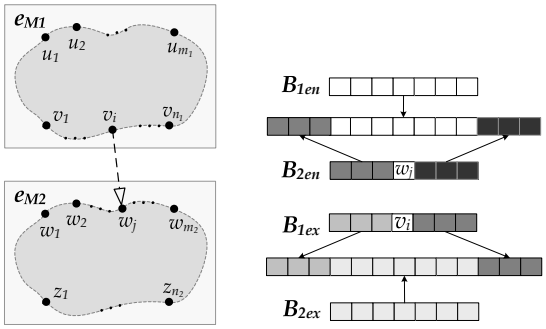


Figure 17: Merge two misc composites e_{M1} and e_{M1} connected by a path (v_i, w_j)

progressively merging misc composites. An element is always merged to another element with ordering.

For effective merging and ordering, we designed the (m,n) misc composite such that it self-manages its structure and order. A misc composite e_M contains (1) a point composite map $\mathbf{E_P}$ in which a set of an arbitrary number of point composites $E_P = E_F \cup E_{TR} \cup E_C$ are indexed independently by their entry vertices, (2) a set of paths P that connect the points, (3) an ordered set of m entry vertices B_{en}, and (4) an ordered set of n exit vertices B_{ex}.

The boundary vertices of points in E_P are either the terminal vertices of the paths or the boundary vertices of the containing misc composite. An element of E_P or E_M can be merged into another element of E_M with ordering. Let $e_{M1} = (\mathbf{E_{1P}}, P_1, B_{1en}, B_{1ex})$ and $e_{M2} = (\mathbf{E_{2P}}, P_2, B_{2en}, B_{2ex})$ be two distinct misc composites, where

- $B_{1en} = (u_1, u_2, \cdots, u_{m_1})$, $B_{1ex} = (v_1, v_2, \cdots, v_{n_1})$;
- $B_{2en} = (w_1, w_2, \cdots, w_{m_2})$, $B_{2ex} = (z_1, z_2, \cdots, v_{n_2})$.

Suppose a path from $v_i \in B_{1ex}$ to $w_j \in B_{2en}$ (Figure 17) is explored during a walk. Then e_{M2} can be merged into e_{M1} with ordering by the MERGE function (Appx. B Alg. 2). Because the two terminal vertices of path (v_i, w_j) are no longer boundary vertices in the merge, they are removed from the boundary vertex sets. Each removal splits the respective entry or exit vertex set into two parts which are joined to the two ends of the corresponding set of the other misc composite[30] (Figure 17). Suppose the path (v_i, w_j) is from a misc composite e_M to a point composite e_P. Then e_P can be merged into e_M by first merging e_P into an empty misc composite (Appx. B Alg. 3). Merging a misc composite into a point composite can be performed in the same manner. In some cases, the path (v_i, w_j) found may start from and end at the same misc composite e_M. Such a path forms a cycle around e_M, i.e., (v_i, w_j) is a backward path (Cormen et al., 2001). The cycle has to be closed (Appx. B Alg. 4).

These merge options combined with the DFS walk of the paths surrounding the point composites make the MCF algorithm (Appx. B Alg. 5). After MCF visits all point composites, the algorithm terminates. We update the hyperedge map $\mathbf{E}$ using the misc composite map $\mathbf{E_M}$: (1) remove from $\mathbf{E}$ the point composites that are merged in to misc composites (i.e., $\forall e_P \in e_M \in \mathbf{E_M}$ with their indexes $\forall u \in B_{en} \in e_P$) , and (2) add to $\mathbf{E}$ all misc composites indexed by their entry vertices (i.e., $\forall e_M \in \mathbf{E_M}$ with their indexes $\forall u \in B_{en} \in e_M$).

Note that there are likely some point composites that *remain* in $\mathbf{E}$. These composites will be transformed individually into model components according to the point types, i.e., e_F, e_{TR} and e_C, in transformation step 3.

The misc composite detection is the last sub-step in transformation step 2. After this, we obtain graph $G_2 = (\mathbf{T}, \mathbf{E})$, in which $\mathbf{T}$ (unchanged as in G_1) is a map of track vertices, and $\mathbf{E}$ is a map of hyperedges each of which represents a composite with a corresponding type (Info. Type 7). The information about point composites (Info. Type 6) is contained in the corresponding hyperedges of the containing composites. The (track, track) edges in the independent paths connecting the hyperedges are represented by entries in $\mathbf{T}$. $\mathbf{T}$ is an ordinary graph that describes vertex-vertex relations at the lowest level of the composition. $\mathbf{E}$ is insofar a sufficient representation of the model composition as a CCH. The transformation from G_0 to *MCG* is completed at $G_2 = (\mathbf{T}, \mathbf{E})$. Each hyperedge type (and the represented composite) has a corresponding model component (type) in the LIBROS library. During the model instantiation in the next step, each hyperedge $e \in \mathbf{E}$ can be transformed into a coupled model component instance according to its type.

4 Model Generation

For this step, we shall design a product p_3 to instantiate a simulation model G_3 according to G_2 which is an *MCG* that contains sufficient information about the model structure and composition. In the light-rail case, G_3 is a `TopLevelModel` composed of LIBROS model components. Recall that a top level model is the root model that is at the highest level of the model composite hierarchy. For model instantiation, besides the information in G_2, we often need other information about model configuration, which includes the setting of the model parameter values and the initial values of model variables. We used additional data sources for LIBROS model configuration[31]. The data shall be prepared such that the quality issues related to syntactic consistency and mapping consistency are dealt with before model instantiation[32]. From now on, we assume that the date is transformed

[30] Note that it preserves the original ordering which numbers the entry or exit vertices counterclockwise.

[31] For model configuration, we used the following data provided by HTM: (1) timetables that schedule the services, (2) the routes of the service lines, (3) vehicle types for the service lines, (4) Service line transformation, i.e., the locations where a service line changes to another.

[32] Common approaches, such as data type and format conversion, data merging and mapping tables, are used for the preparation. The implementations of the approaches are domain specific. Therefore we will not discuss the details.

into appropriate unit and structure, and it is indexed with identifiers that are consistent with the identifiers[33] contained in G_2.

4.1 Model Instantiation

The instantiation (or generation)[34] of a simulation model in this step can be completed with one DFS graph traversal of $G_2 = (\mathbf{T}, \mathbf{E})$. The traversal, however, does not walk deeper into the hyperedges $e \in \mathbf{E}$ except for the boundary vertices. Once reached an entry vertex of a hyperedge, the walk "exits" the hyperedge and walks further as usual. In this sense, the hyperedges are treated like vertices in the walk. Along the walk, model component instances are generated and configured. Each vertex or hyperedge is transformed into a model component according to its type. Each hyperedge corresponds to a coupled model. The ordinary edges, i.e., the (vertex, vertex) pairs in **T**, indicate the coupling relations between the model components. Model configuration in our case takes place only at the elementary model level (i.e., in the atomic models).

4.1.1 Model Instantiation Basics

A vertex in **T** is transformed into a corresponding model component instance. In LIBROS, it can be a `Source` or a `TrackSegment`. An atomic model component in LIBROS is called a *Rail Infrastructure Element* (RIE). The other RIEs are not directly represented by individual vertices in **T**. The sensors, switches and signals (3S) models, e.g, are represented by the vertex combinations that are contained in the hyperedges in **E**. They are therefore generated when the hyperedges are generated (Section 4.1.2); so are the control units. The RIE models are defined with fixed structures (including port settings). Their instantiation is straightforward. Each model instantiation shall specify the parent model of that model. For example, when G_2 is transformed into a `TopLevelModel` M_{top}, any model that is placed directly under the `TopLevelModel` has M_{top} as the parent. A modes is configured after instantiation. In addition, for the purpose of animation, a model image is generated for each (atomic) model instance that is to be animated. After the model generation, the images are passed on to the model image manager which is an animation component in the LIBROS library[35].

4.1.2 Model Instantiation from A Hyperedge

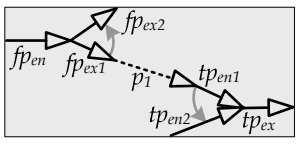
A hyperedge in **E** has specified constituent composites, paths and boundary vertices. The transformation does not "traverse" the hyperedge but instantiates the corresponding model according to the hyperedge in a particular manner. Since many of the composites are intersections, we use a simple misc composite to explain how a coupled model is instantiated from a hyperedge by the *Infrastructure Component Generation* (ICG) algorithm.

Note that the models generated are all *ordered* according to the ordering in the composites. In particular this means that the start nodes and end nodes (for coupling) of each model are ordered respectively so that they can be matched with the ordered composites.

Example 1 *Transform a* $(2,2)$ *misc composite* e_M *into an infrastructure model* M*. Let* e_M *(and its specification) be as shown in Figure 18.*

Given the $(2,2)$-edge e_M, the ICG first creates a (coupled) model M of type `MiscCrossing` (an infrastrcture component in LIBROS); see Figure 19 (A): (1) M is initialized with two start nodes SN_1, SN_2 and two end nodes EN_1, EN_2. (2) Two lineside signals LS_1, LS_2 are added, one at each entry (i.e., start node) of the intersection. (3) A facing point FP and a trailing point TP are added to M. (4) A control unit CU is added to M, with which LS_1, LS_2, FP and TP are coupled. For the convenience of infrastructure coupling, a queue like structure *Que* is used to hold successive $(1,1)$ infrastructure models (which together form an independent path) so that they can be later coupled together at once. In a coupled infrastructure model I_C, each start node of I_C and each end node of a non-$(1,1)$ constituent component in I_C is associated with a *Que*. For example, there are five *Que*s in M: two for its two start nodes, two for the two end nodes of FP, and one for the end nodes of TP. LS_1 is added to Que_{SN1}, and LS_2 is added to Que_{SN2} since signals are to guard the inflow traffic of the intersection.

Next, the ICG generates one track segment for each entry vertex of the hyperedge e_M, and adds them into the corresponding start nodes' *Que*s. If an entry vertex of e_M is not an entry vertex of a point component in e_M but holds a path to the latter, then all (models of the) vertices in the path[36] (including the target vertex) are added into the corresponding *Que* (Info. Type 8). (Hence the start nodes' *Que*s are also called entry path *Que*s.) To end this step, the (first) start node and the (last) end node (of the models) of each entry path *Que* are coupled to the corresponding attachment nodes. For example, Figure 19 (B), since fp_{en} is the first entry vertex of e_M (and the entry vertex of fp), a `TrackSegment` T_1 is generated and added to Que_{SN1}. The start node of LS_1 (the first element in Que_{SN1})



Composites		fp, tp
Paths	p_1	fp_{ex1} to tp_{en1}
Entry	1. e_{Men1}	tp_{en2}
	2. e_{Men2}	fp_{en}
Exit	1. e_{Mex1}	tp_{ex}
	2. e_{Mex2}	fp_{ex2}

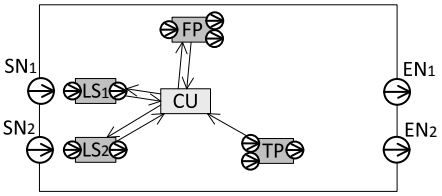
Figure 18: A $(2,2)$ misc component e_M

[33] Two types of identifiers appear in G_2: the identifiers for stops and the identifiers for switches.

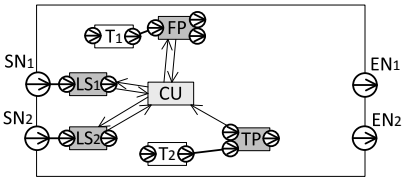
[34] We use instantiation and generation interchangeably hereinafter since they mean the same in the context of transformation step 3.

[35] Readers of interest may refer to Huang (2013, Chapter 4).

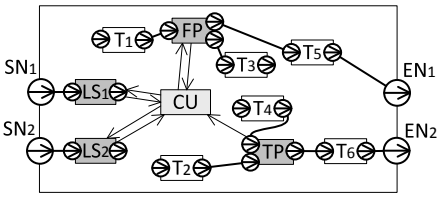
[36] The terminal vertices of the path are specified in the hyperedge. The vertices between them are in **T**.



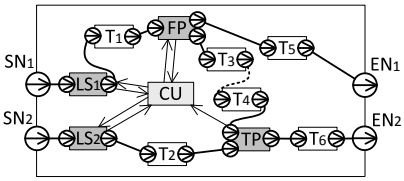
(A) Initialization with start and end nodes; add points, lineside signals and a control unit. *Que* status: $Que_{SN1} = (LS_1)$, $Que_{SN2} = (LS_2)$



(B) Generate track segments in the entry paths and add to entry path *Que*s; couple the start and end nodes of each entry path *Que* to their corresponding attachment nodes. *Que* status: $Que_{SN1} = (LS_1, T_1)$, $Que_{SN2} = (LS_2, T_2)$



(C) Generate track segments in the internal and exit paths and add to *Que*s; couple the start and end nodes of each path *Que*. *Que* status: $Que_{SN1} = (LS_1, T_1)$, $Que_{SN2} = (LS_2, T_2)$, $Que_{FP1} = (T_3, \cdots, T_4)$, $Que_{FP2} = (T_5)$, $Que_{TP} = (T_6)$



(D) Couple the models in the *Que*s. *Que* status: the same as (C).

Figure 19: Steps of generating a coupled infrastructure model M – Example 1 (cf., Figure 18)

and the end node of T_1 (the last element in Que_{SN1}) are coupled to SN_1 and the start node of *FP* correspondingly.

The next step, Figure 19 (C), is similar to the previous one, but the ICG generates one track segment for each exit vertex of the point composites of the hyperedge e_M and adds them to the corresponding *Que*s. Since the exist vertex may hold a path to an entry vertex of another point composite, or to an exit vertex of e_M (or be an exist vertex of e_M itself), these *Que*s are also called internal path *Que*s or exit path *Que*s correspondingly. After these *Que*s are completed, their start and end nodes are coupled to the corresponding attachment nodes as in the previous step.

In the last step, the $(1,1)$ infrastructure models in each (entry, internal or exit path) *Que* are coupled sequentially to one another, i.e., the end node of a previous model is coupled to the start node of a next model. The infrastructure model generation of the hyperedge completes at this step; Figure 19 (D).

Note that the shaded model components in Figure 19 (i.e., the lineside signals, points, and control unit) do not have corresponding vertices in the hyperedge. The information contained in the hyperedge is sufficient to indicate the composition of the coupled infrastructure model including these extra model components, by which we obtain Info. Type 11. This means that the hyperedge (composite) is injective homomorphic to the coupled infrastructure model. The four-step model generation is applicable to all intersection hyperedge composites.

4.1.3 Configuration of Control Unit

Each intersection model component in LIBROS contains a control unit (CU). The CU interacts with the detectors and signals in the intersection and supervises the safe operation of the area. In this regard, a CU maintains a check table that contains a list of all possible routes in the intersection, the required points (and positions if applicable) by the routes, the states (whether reserved and/or occupied) of the points, weather the routes are active or have queueing requests, etc. Since the layout of a misc crossing is not known a priori, we need an algorithm that can configure the routes in the check table (the first two types of information) automatically according to the layout[37]. Otherwise, the generation of misc crossings is of little use.

Find All Routes in An Intersection A route, denoted by $r = (s, \rho_1, \rho_2, \cdots, \rho_i)$, is composed of an entrance signal and the required points (and positions if applicable) along the route in the intersection. For example, the model M (Figure 19) generated in Example 1 has three routes, which are $r_1 = (LS_1, FP/2)$, $r_2 = (LS_1, FP/1, TP)$, $r_3 = (LS_2, TP)$. The number following *FP* is the position (i.e., ordering) of the facing point required by the route. Because all possible routes in an intersection are "rooted" from the entries of the intersection, this finding-all-routes problem is essentially a finding-all-paths problem with given start vertices in a directed acyclic graph. We again use DFS for the route finding, but a search does not stop at a visited vertex. We use the *Que*s as "edges" in the search since they represent the entry, internal and exit paths in the intersection.

A search starts at an entry path *Que* whose first element (which is a lineside signal) is added to a new route, i.e., $r = (s)$. The point that is coupled with the end of the *Que* is added into r, e.g., $r = (s, p_1)$. The search walks deeper to an end node *Que* of the point. In case of a trailing point (TP) or crossing point (CP), the route proceeds only in one direction. But in case of a facing point (FP),

[37] The algorithm also works on intersections with fixed layout, i.e., those match the composite patterns (Section 3.3.2).

the original route becomes two routes, and the order of the end nodes in the FP are added into the corresponding route, e.g., $r = (s, p_1/1)$, $r' = (s, p_1/2)$. The search can walk deeper to either end node *Que* with the corresponding route record. The route search continues in the same manner until it reaches an exit path *Que*, and when all the entry path *Que*s are explored.

Assign Routes to Service Lines Each service line has a preassigned service route (from terminal to terminal). The route data specifies the direction of each line at each facing point. This information needs to be transformed into the corresponding route of the intersection since the vehicle models would request the corresponding route in the simulation. After this route configuration for service lines, the configuration of a control unit is complete.

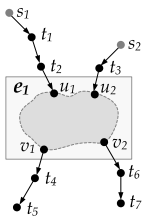
4.1.4 Model Instantiation and Setting Up Couplings

Setting up couplings among models is not as straightforward as it may appear. A coupling relation is a port-to-port relation. Since a model may have more than one start or end nodes, coupling relations often can not be directly translated from model-to-model relations. In Example 1, when `TrackSegment` T_3 or T_5 is to be coupled to *FP*, we need to know the corresponding end node of *FP*. This can be known, e.g., by keeping the order of the end nodes and of the to be coupled `TrackSegment`s.

Additionally, G_2 is injective homomorphic to the generated model G_3 such that a vertex-to-vertex relation in G_2 may represent an indirect model-to-model relation in G_3. This is the case when only one of two connected vertices is a boundary vertex of a hyperedge while the other vertex is not in that hyperedge, so that only one is transformed to be a component of a higher level model. The two corresponding models (of the vertices) as such can not be directly coupled to each other.

Example 2 *Consider a G_2, Figure 20, with two sources s_1, s_2, one $(2,2)$-edge e_1, and a number of track vertices.*

Suppose the DFS starts at s_1. It shall first walk through $s_1 \to t_1 \to t_2$ because these vertices have entries in **T** but not in **E**. At each step, a model is generated, say S_1, T_1, T_2, and since they do not have multiple start or end nodes, we can successively couple them. At step (t_2, u_1), u_1 has an entry (u_1, e_1) in **E** (i.e., u_1 is an entry vertex of hyperedge e_1), which can be e.g., a $(2,2)$ diamond composite or a misc composite, hence a coupled model, say M_1, is generated according to e_1 (Section 4.1.2). Note that M_1 is the parent model of models of u_1, u_2, v_1, v_2, i.e., U_1, U_2, V_1, V_2 which are all generated at once by generating M_1.



$$\mathbf{T} = \{(s_1,t_1),(t_1,t_2),(t_2,u_1),(s_2,t_3),(t_3,u_2),(v_1,t_4),(t_4,t_5),(v_2,t_6),(t_6,t_7),\cdots\}$$
$$\mathbf{E} = \{(u_1,e_1),(u_2,e_1)\}$$

Figure 20: A simple G_2 with two sources s_1, s_2 and one $(2,2)$-edge e_1

At this point, we need to "couple T_2 to U_1" through a start node of M_1 instead of coupling them directly together. Suppose M_1 is a $(2,2)$ `MiscCrossing` as shown in Example 1 (Figure 19 D). Whether T_2 shall be coupled to EN_1 or EN_2 depends on which node U_1 is "coupled to" (through a lineside signal). This information can be known, e.g., through the order of the *Que* that contains U_1[38]. Similarly, when the walk "exists" e_1 and goes to, e.g., $v_1 \to t_4 \to t_5$, we use the order of the *Que* that contains V_1 to know the order of the end node of M_1 that shall couple to T_4. Because t_5 is the end of the branch (i.e., it does not have an entry in **T**), we generate a sink model (where a vehicle model is removed during a simulation run) SK_1 and couple it to T_5.

The DFS then starts at the second source s_2 and continues with $s_2 \to t_3 \to u_2$. Because there exits an entry (u_2, e_1) in **E** and a model M_1 of e_1 is already generated, we only need to couple T_3 to the corresponding start node of M_1 (as discussed). The walk then goes on to $v_2 \to t_6 \to t_7$, couple T_6 to the corresponding end node of M_1, and ends by generating SK_2 after T_7.

Model Map Through traversing a simple example of G_2, we can also observe that some mapping is needed to maintain a relation between G_2 and G_3 in order to trace back which part of G_2 has been transformed into which part of G_3. A DFS in G_2 anyway needs some record to trace the visited vertices. But the model generation (and couplings) also needs a mapping between the vertices (and hyperedges) and the generated models. For example, in traversing the G_2 in Figure 20, at step (t_3, u_2), we need to know the counterpart of u_2 in G_3 or whether the model is generated at all in order to have further information about the parent model and the corresponding start port. We therefore use a *model map* **M** of (vertex, model) pairs for the mapping between the vertices in G_2 and their direct counterpart models (not parent models) in G_3. Since the source vertices do not have "back walks", they are not recorded in **M**. Hence **M** has only (track vertex, `TrackSegment`) pairs. These pairs are added into **M** each time a track vertex (including the ones in the hyperedges) is visited and the corresponding `TrackSegment` is generated. The information about the hyperedges (and their corresponding models) can be obtained through the entry vertices.

[38] When the nodes are ordered, the *Que*s are ordered too.

4.2 Transformation Step 3: Generation of Simulation Model

Through transformation step 2, the CCH (i.e., the hyperedge map **E**) of G_2 is defined bottom-up on the ordinary digraph (i.e., the track map **T**). Given $G_2 = (\mathbf{T}, \mathbf{E})$, we use a *Model Generation Algorithm* (MGA), see Appx. C Alg. 7, in transformation step 3 to generate G_3, a simulation model. The principle of the algorithm is explained by the two examples in Section 4.1. In this step, the model generation of G_3 is partly bottom-up, as when the graph traversal is at ordinary vertices of **T**, and partly top-down, as when the traversal encounters a hyperedge in **E** based on which an `InfraComponent` is generated.

5 Model Generator

In LIBROS, there is a `ModelGeneator` component that implements the steps of model transformation (steps 1 and 2) and instantiation (step 3) as discussed in Section 3 and Section 4. The main tasks performed by the model generator (Figure 21) can be summarized as following.

1. Read CAD data into G_0[39]. The CAD entities are the vertices in G_0. The entity types and descriptions are the vertex types and attributes.
2. Transformation step 1. Build a directed graph $G_1 = (\mathbf{T}, \mathbf{P_F}, \mathbf{P_T}, \mathbf{P_C}, \mathbf{E})$ where track compositions of points (in $\mathbf{P_F}, \mathbf{P_T}, \mathbf{P_C}$) and stop composites e_S (in **E**) are detected: (a) Partition the vertices into three sets: sources, tracks and stops. (b)"Traverse" G_0 based on the geometrical connectedness of the vertices in order to create directed edges (in **T**), detect locations of points and create stop hyperedges.
3. Transformation step 2. Build an *MCG* $G_2 = (\mathbf{T}, \mathbf{E})$ where **T** remains the same as in G_1 and **E** contains more infrastructure composites as hyperedges. The transformation starts with G_1 and rewrites it incrementally with sub-steps based on the result of incremental pattern matching. These sub-steps are partially ordered. The composite matching or finding algorithm in a sub-step does not traverse the whole graph but walks the independent paths surrounding the candidate constituent composites.
4. Read in the other model configuration data such as timetables and service routes.
5. Transformation step 3. Generate a `TopLevelModel` model G_3 by traversing G_2 and creating infrastructure model components according to the vertex and hyperedge types each of which has a corresponding model component (type) in LIBROS. The components are configured and coupled accordingly. Model images are as well generated for the purpose of animation.

After these steps, we obtain a simulation model G_3 where the model behavior at the elementary level is prespecified in the (atomic) model components in the LIBROS library, and the model structure is dynamically constructed using the coupled components according to the infrastructure CAD data.

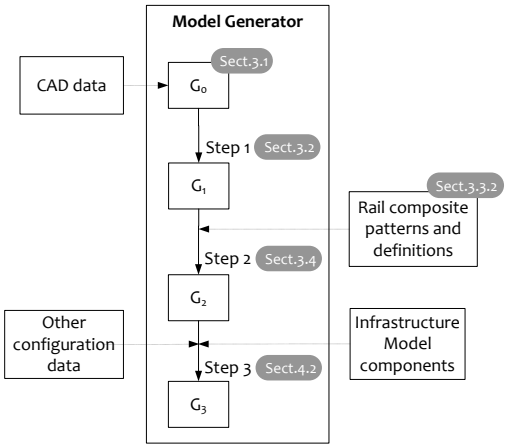


Figure 21: Model generator

6 Conclusions

In this paper, we presented our study on *Automated Model Generation* (AMG). A method is developed that can automatically generate complex simulation models from existing data using model components as building blocks. The AMG method differs from other methods in that: the data used for the AMG does not contain specifications of the model structures to be generated; and the generated models have structures that are dynamically constructed. To study the AMG method, we used cases in the domain of light-rail transport. In generating LIBROS models, infrastructure CAD data is used as a basis for model structures. The major challenge for the AMG method is the presentation suitability of model structures (including identification of model components and compositional relations). In principle, transformation rules for AMG can be defined when the data that serves as input for AMG has semantic and pragmatic completeness, has definable measures for syntactic and mapping inconsistency (if any), and when modelers have sufficient domain knowledge and deductive reasoning for the definition of transformation rules that can solve data issues related to semantic accuracy and presentation suitability with regard to model structure and parameterization. Model transformations are defined on meta-models of the original data structure (as the AMG input), of intermediate structures and of the simulation model. Suppose that the original data structure is a non-graph, meaning that the data items do not have specified relations (or they can be deemed as a graph with vertices but without

[39] A Java Ycad library (sourceforge.net/projects/ycad) is used to read the AutoCAD DXF files.

edges); and assume that the meta-model of the simulation model and the model components are specified, e.g., in a domain model component library, then the transformation needs three steps: from a non-graph to a digraph, to a model composite graph, and finally to a simulation model, where the vertices and hyperedges of the model composite graph must have corresponding (pre-specified) model components.

To conclude, the functionality that the AMG method we propose should provide transformation rules: (1) that can solve data quality issues related to semantic accuracy and presentation suitability in terms of model structure and parameterization, (2) that are defined on the meta-models of the original data structure, of the indeterminate structures, and of the simulation model, (3) that uses graph and/or hypergraph-based structure representation and transformation, (4) that can construct a representation of the (to be generated) model structure whose components can be mapped to corresponding pre-specified simulation model components, (5) that can construct a simulation model according to the representation of the model structure. The following are not necessary but can be beneficial for the design of the transformation rules: (1) recursive definition and incremental search of model composite patterns, and (2) using both composition and decomposition in the transformation rules.

To complement the AMG method, when operational data from the real system is available, the simulation model can be calibrated by a model calibration procedure using user defined goodness-of-fit measures and parameter search algorithms. Note that the goodness-of-fit measures in the calibration procedure can serve as a way to validate the relevant model output data (operational validation through comparison). The AMG method is evaluated with a panel of subject-matter experts and it has practical uses that helped strategic, tactical and operational decision makings in light-rail transport systems[40].

There are limitations that are bound to such model generation processes. The transformation rules (as well as the data patterns and simulation model components) are designed for certain types of input data with the assumptions that the data satisfies certain properties. When such properties change or when those assumptions are invalid, the AMG rules are likely to be ineffective and/or would cause errors. Even if the rules are defined flexible enough to accommodate large degrees of freedom, there always can be changes in data that are out of the scope that was under consideration during the process design. At the very end, it is a trade-off between complexity (of the rules and development) and flexibility. Moreover, since the transformation rules are based on graph theory, the data and model need to be represented with graph-based structures. This is not necessarily always the case. The main contribution of the paper is the AMG method or the process. It is participially interesting for those who need to frequently develop different models for large systems where there are reoccurring component parts and definable rules. If modelers want to invest time and effort to develop an AMG routine, then the method we proposed is a possible solution.

With regard to future research, multi-resolution and multi-perspective model generation can be an interesting and challenging direction. The development of a domain simulation model or a class of models is often intended for similar purposes of study. When simulation models can be generated with different resolutions and/or perspectives from a library or a set of libraries, such libraries will be highly reusable. This of course requires substantial research efforts. Above all, we need libraries that contain sufficient domain knowledge and organize different parts of this knowledge in an appropriate manner to allow users or automated agents to query the knowledge with relevance. Furthermore, the AMG method designed in this research can be applied for AMG of other systems besides light-rail transport systems. Straightforward applications include those in the infrastructure domain such as heavy-rail and road transport, pipeline and grid systems. A step further could be made to systems that can be represented with graph-based structures such as production and supply chain systems.

[40] Details about model validation and applications can be found in Huang (2013, Chapter 7 and 8).

Acknowledgement

The authors thank Martijn Warnier from TU Delft for his feedback.

References

AMS (2013). *Simulation stimulates Ford's improvement*. Automobile Manufacturing Solutions.

Amsterdam, J. (1993). "Automated Qualitative Modeling of Dynamic Physical Systems". PhD thesis. Artifical Intelligence Laboratory, Massachusetts Institue of Technology.

Andersson, P. (2006). *Hyperedge Replacement Grammars*.

Banks, J., J. S. Carson II, B. L. Nelson, and D. M. Nicol (2010). *Discrete-Event System Simulation*. 5th. Pearson Education.

Batini, C. and M. Scannapieco (2006). *Data Quality: Concepts, Methodologies and Techniques*. Data-Centric Systems and Applications. Springer-Verlag Berlin Heidelberg.

Ben-Ari, M. (2012). *Mathematical Logic for Computer Science*. Springer-Verlag London.

Berge, C. (1973). *Graphs and Hypergraphs*. Translation and revised edition of Graphes et Hypergraphes 1970. North-Holland Publishing Company.

Berge, C. (1989). *Hypergraphs: Combinatorics of Finite Sets*. Vol. 45. North-Holland Mathematical Library. Elsevier Science Publishers.

Bergmann, S. and S. Strassburger (2010). "Challenges for the Automatic Generation of Simulation Models for Production Systems". In: *Proceedings of the 2010 Summer Simulation Multiconference*. Ottawa, Canada, pp. 545–549.

Brause, R. (2004). "Model selection and adaptation for biochemical pathways". In: *Lecture Notes in Computer Science* 3337, pp. 439–449.

Bruni, R., F. Gadducci, and A. Lluch Lafuente (2010). "An Algebra of Hierarchical Graphs". In: *Trustworthly Global Computing*. Ed. by M. Wirsing, M. Hofmann, and A. Rauschmayer. Vol. 6084. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 205 –221.

Busatto, G. and B. Hoffmann (2001). "Comparing notions of hierarchical graph transformation". In: *Electronic Notes in Theoretical Computer Science* 50.3, pp. 310 –317.

COBP (2002). *NATO code of best practice for command and control assessment*. DoD Command and Control Research Program (CCRP). SAS-026.

Cai, J. (2011). "Assessing The Impact of Capacity of Depots and Vehicle Schedule in Transportation Systems". MA thesis. Delft University Of Technology, Faculty Of Technology, Policy And Management.

Cao, Y., Y. Liu, H. Fan, and B. Fan (2012). "SysML-based uniform behavior modeling and automated mapping of design and simulation model for complex mechatronics". In: *CAD Computer Aided Design*. Article in Press.

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein (2001). *Introduction to Algorithms*. 3rd. MIT Press and McGraw-Hill.

Corradini, A., U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe (1997). "Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach". In: *Handbook of Graph Grammars and computing by graph transformation*. Ed. by G. Rozenberg. Vol. 1: Foundations. World Scientific Publishing, pp. 163 –246.

Crosbie, R. E. (2010). "Grand Challenges in Modeling and Simulation". In: *SCS M&S Magazine* 1.1, pp. 1–8.

Czarnecki, K. and S. Helsen (2006). "Feature-based survey of model transformation approaches". In: *IBM Systems Journal* 45.3, pp. 621 –645.

De Lara, J., H. Vangheluwe, and M. Alfonseca (2004). "Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3". In: *Software and Systems Modeling* 3.3, pp. 194 –209.

Drewes, F., H.-J. Kreowski, and A. Habel (1997). "Hyperedge Replacement Graph Grammars". In: *Handbook of Graph Grammars and Computing by Graph Transformation*. Ed. by G. Rozenberg. Vol. 1: Foundations. World Scientific Publishing, pp. 95 –162.

Drewes, F., B. Hoffmann, and D. Plump (2002). "Hierarchical Graph Transformation". In: *Journal of Computer and System Sciences* 64.2, pp. 249 –283.

Eeckelaert, T., W. Daems, G. Gielen, and W. Sansen (2004). "Generalized simulation-based posynomial model generation for analog integrated circuits". In: *Analog Integrated Circuits and Signal Processing* 40.3, pp. 193–203.

Ehrig, H., K. Ehrig, U. Prange, and G. Taentzer (2006a). "Fundamental Theory for Typed Attributed Graphs and Graph Transformation based on Adhesive HLR Categories". In: *Fundamenta Informaticae* 74.1, pp. 31 –61.

Ehrig, H., K. Ehrig, U. Prange, and G. Taentzer (2006b). *Fundamentals of Algebraic Graph Transformation*. Ed. by W. Brauer, G. Rozenberg, and A. Salomaa. Monographs in Theoretical Computer Science, An EATCS Series. Springer-Verlag Berlin Heidelberg.

Fan, W., J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu (2010). "Graph pattern matching: from intractable to polynomial time". In: *Proceedings of the VLDB Endowment* 3.1-2, pp. 264 –275.

Ferney, M. (2000). "Modelling and controlling product manufacturing systems using bond-graphs and state equations: Continuous systems and discrete systems which can be represented by continuous models". In: *Production Planning and Control* 11.1, pp. 7–19.

Fowler, J. W. and O. Rose (2004). "Grand Challenges in Modeling and Simulation of Complex Manufacturing Systems". In: *Simulation* 80.9, pp. 469–476.

Gallagher, B. (2006). "Matching structure and semantics: A survey on graph-based pattern matching". In: *Capturing and Using Patterns for Evidence Detection*. AAAI Press, pp. 45–53.

Gelsey, A. (1990). "Automated reasoning about machines". PhD thesis. New Haven: Yale University.

Gelsey, A. (1995). "Automated reasoning about machines". In: *Artificial Intelligence* 74.1, pp. 1–53.

Glotzer, S. C., S. Kim, P. T. Cummings, A. Deshmukh, M. Head-Gordon, G. Karniadakis, L. Petzold, C. Sagui, and M. Shinozuka (2010). *International Assessment of Research and Development in Simulation-based Engineering and Science*. Tech. rep. World Technology Evaluation Center.

Gössler, G. and J. Sifakis (2005). "Composition for component-based modeling". In: *Science of Computer Programming* 55.1-3, pp. 161–183.

Granda, J. J. and R. C. Montgomery (2003). *Automated Modeling and Simulation Using the Bond Graph Method for the Aerospace Industry*. Tech. rep. CASI-20040085773. NASA Langley Research Center.

Habel, A. (1992). *Hyperedge Replacement: Grammars and Languages*. Spriger-Verlag.

Harrison, G. A., D. S. Maynard, and E. Pollak (2004). "Automated database and schema-based data interchange for modeling and simulation". In: *Proceedings of the 2004 Winter simulation Conference*, pp. 191–197.

Huang, Y., A. Verbraeck, N. van Oort, and H. Veldhoen (2010). "Rail Transit Network Design Supported by an Open Source Simulation Library: Towards Reliability Improvement". In: *Transportation Research Board 89th Annual Meeting Compendium of Papers*. 10-0310. Washington, DC, USA: TRB.

Huang, Y. (2013). "Automated Simulation Model Generation". PhD thesis. Delft Univeristy of Technology.

Jiang, X. Y. and H. Bunke (1996). "Including geometry in graph representations: A quadratic-time graph isomorphism algorithm and its applications". In: *Advances in Structural and Syntactical Pattern Recognition*. Ed. by P. Perner, P. Wang, and A. Rosenfeld. Vol. 1121. LNCS. Springer Berlin Heidelberg, pp. 110 –119.

Jiang, X. Y. and H. Bunke (1999). "Optimal quadratic-time isomorphism of ordered graphs". In: *Pattern Recognition* 32.7, pp. 1273 –1283.

Johnson, T., A. Kerzhner, C. Paredis, and R. Burkhart (2012). "Integrating models and simulations of continuous dynamics into SysML". In: *Journal of Computing and Information Science in Engineering* 12.1.

Kahl, W. (2002). *A Relation-Algebraic Approach to Graph Structure Transformation*. Habilitationsschrift.

Kamerling, W. (2007). "Besluitvorming over traminfrastructuur". In Dutch. MA thesis. Delft Univeristy of Technology, Faculty of Technology, Policy and Management.

Kanacilo, E. M. and A. Verbraeck (2006). "Simulation services to support the control design of rail infrastructures". In: *Proceedings of the 2006 Winter Simulation Conference*. IEEE, pp. 1372–1379.

Kanacilo, E. M. and A. Verbraeck (2007). "Assessing tram schedules using a library of simulation components". In: *Proceedings of the 2007 Winter Simulation Conference*. IEEE, pp. 1878–1886.

Kleppe, A., J. Warmer, and W. Bast (2003). *MDA Explained: The Model-Driven Architecture: Practice and Promise*. Addison Wesley.

Levy, A. Y., Y. Iwasaki, and R. Fikes (1997). "Automated model selection for simulation based on relevance reasoning". In: *Artificial Intelligence* 96.2, pp. 351–394.

Little, S., D. Walter, K. Jones, C. Myers, and A. Sen (2010). "Analog/mixed-signal circuit verification using models generated from simulation traces". In: *International Journal of Foundations of Computer Science* 21.2, pp. 191–210.

Longo, F. (2011). "Advances of modeling and simulation in supply chain and industry". In: *Simulation* 87.8, pp. 651–656.

Lucko, G., P. C. Benjamin, K. Swaminathan, and M. G. Madden (2010). "Comparison of manual and automated simulation generation approaches and their use for construction applications". In: *Proceedings of the 2010 Winter Simulation Conference*, pp. 3132–3144.

Manin, Y. I. (2010). *A Course in Mathematical Logic for Mathematicians*. Vol. 53. Graduate Texts in Mathematics. Springer New York.

Manning, C. D., H. Schütze, and P. Raghavan (2008). *Introduction to Information Retrieval*. Cambridge University Press.

Mens, T. and P. van Gorp (2006). "A Taxonomy of Model Transformation". In: *Electronic Notes in Theoretical Computer Science* 152, pp. 125–142.

Mielczarek, B. and J. Uzialko-Mydlikowska (2012). "Application of computer simulation modeling in the health care sector: a survey". In: *Simulation* 88.2, pp. 197–216.

Mueller, R. (2007). "Specification And Automatic Generation Of Simulation Models With Applications In Semiconductor Manufacturing". PhD thesis. Georgia Institute of Technology.

Nayak, P. (1995). *Automated Modeling of Physical Systems*. Vol. 1003. Lecture Notes in Computer Science.

Pachl, J. (2002). *Railway Operation and Control*. VTD Rail Publishing.

Palacz, W. (2004). "Algebraic hierarchical graph transformation". In: *Journal of Computer and System Sciences* 68.3, pp. 497 – 520.

Roman, M. and D. Selisteanu (2012). "Pseudo bond graph modeling of wastewater treatment bioprocesses". In: *Simulation* 88.2, pp. 233–251.

Roychoudhury, I., M. Daigle, G. Biswas, and X. Koutsoukos (2011). "Efficient simulation of hybrid systems: A hybrid bond graph approach". In: *Simulation* 87.6, pp. 467–498.

Shanks, G. and B. Corbitt (1999). "Understanding Data Quality: Social and Cultural Aspects". In: *Proceeding of the 10th Australasian Conference on Information Systems*, pp. 785 –797.

Shannon, R. E. (1975). *Systems simulation: the art and science*. Prentice Hall, Inc.

Theeg, G. and S. Vlasenko, eds. (2009). *Railway Signalling & Interlocking: International Compendium*. Eurailpress.

Thomaseth, K. (2003). "Multidisciplinary modelling of biomedical systems". In: *Computer Methods and Programs in Biomedicine* 71.3, pp. 189–201.

Tian, Y., B. Liu, H.-W. Gao, and W.-Q. Li (2012). "Modeling and simulation of electro-hydraulic proportional position control system with the flexible hose". In: *Advanced Materials Research* 468-471, pp. 2094–2099.

Umesh Rai, B. and L. Umanand (2009). "Bond graph toolbox for handling complex variable". In: *IET Control Theory and Applications* 3.5, pp. 551–560.

Van Oort, N. (2011). "Service Reliability and Urban Public Transport Design". PhD thesis. Netherlands: Delft University of Technology, Department of Transport and Planning.

Vangheluwe, H. (2000). "DEVS as a common denominator for multi-formalism hybrid systems modelling". In: *IEEE International Symposium on Computer-Aided Control System Design*, pp. 129 –134.

Vangheluwe, H. (2008). "Foundations of Modelling and Simulation of Complex Systems". In: *Electronic Communications of the EASST - Proceedings of the 7th International Workshop on Graph Transformation and Visual Modeling Techniques*. 10.

Veldhoen, H. (2009). "Embedding Simulation in Decision Making". MA thesis. Delft University of Technology, Faculty of Technology, Policy and Management.

Velegrakis, Y., J. Miller, and L. Popa (2004). "Preserving mapping consistency under schema changes". In: *The VLDB Journal* 13.3, pp. 274 –293.

Vuchic, V. R. (2005). *Urban Transit: Operations, Planning, and Economics*. John Wiley & Sons, Inc.

Wainer, G. A. (2009). *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. Computational Analysis, Synthesis, and Design of Dynamic Systems. CRC Press.

Wand, Y. and R. Y. Wang (1996). "Anchoring data quality dimensions in ontological foundations". In: *Communications of the ACM* 39.11, pp. 86 –95.

Wang, J., Q. Chang, G. Xiao, N. Wang, and S. Li (2011). "Data driven production modeling and simulation of complex automobile general assembly plant". In: *Computers in Industry* 62.7, pp. 765–775.

Wasynczuk, O. and S. Sudhoff (1996). "Automated state model generation algorithm for power circuits and systems". In: *IEEE Transactions on Power Systems* 11.4, pp. 1951–1956.

Weisstein, E. W. (2009). *Graph Automorphism*. MathWorld - A Wolfram Web Resource.

Wieland, F. and A. Pritchett (2007). "Looking into the Future of Air Transportation Modeling and Simulation: A Grand Challenge". In: *Simulation* 83.5, pp. 373–384.

Xia, S. and N. Smith (1996). "Automated modelling: A discussion and review". In: *Knowledge Engineering Review* 11.2, pp. 137–160.

Zeigler, B. P., H. Praehofer, and T. G Kim (2000). *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. 2nd. Elsevier/Academic Press.

Zupančič, B. and A. Sodja (2013). "Computer-aided physical multi-domain modelling: Some experiences from education and industrial applications". In: *Simulation Modelling Practice and Theory* 33, pp. 45–67.

Appendixes

A Composite Pattern Matching Algorithm

For the Composite Pattern Matching (CPM) algorithm (Alg. 1), since $P \in \varrho$ is topologically sorted and the non-match of a path excludes the match of the whole pattern, the vertices that satisfy the conditions of the start vertex ex_1 of the first path p_1 are chosen as candidates to start each pattern search. These candidates are saved in T for iteration (ln. 3)[41]. A candidate vertex t_j is used as the start vertex s of the first search path $p_1 \in P$ (lns. 6, 7).

As mentioned in Section 3.3.2, the internal vertices of an independent path must satisfy $d^+ = d^- = 1$. In our case, this means that the internal vertices shall *not* appear in the composite maps but only in $\mathbf{T}$.

Leaving the search bound $\boldsymbol{b}$ aside, a sufficient condition to consider a vertex $\hat{t}$ to be in a path (including the target vertex) is that it is indexed by an known vertex t in the path (lns. 11, 12). If in addition $\hat{t}$ is not in the composite maps, then we can accumulate the search distance d and continue the path walk (ln. 28).

If a known vertex t has no entry in $\mathbf{T}$, then t is literally at the end of the path (with nothing connected to it). In this case, and of course also when the path distance exceeds $\boldsymbol{b}$, the algorithm drops the current search and proceeds with the next candidate $t_j \in T$ (ln. 32).

In walking an independent path p_i given a start vertex s (lns. 11∼30), once a following vertex t appears in the composite maps, it means that there would be a match of p_i if t satisfies the target vertex en_i specification of p_i (i.e., the first two conditions in ln. 14). However, this is *not* necessarily an isomorphism. We need to check whether the found composite $c(t)$, that contains the target vertex t of the (path) image, is pairwise distinct with the other found composites (which are recorded in C'). Note that $C \in \varrho$ is pairwise distinct, hence we have the third condition $c(t) \notin C' \backslash c'_b$ in ln. 14.

If one of the three conditions above is not satisfied (i.e., a non-match) then we can proceed with the next $t_j \in T$ (ln. 24). If these is an isomorphic path match of p_i, the composite $c(t)$ is recorded in C' at the same position as its counterpart in $C \in \varrho$ (ln. 15; note that the index of $c'_b \leftarrow c(t)$ is b as that of $c_b \in C$). In the second case, the algorithm shall proceed with the next path p_{i+1} if there is any path left. Because $P \in \varrho$ is topologically sorted, the composite that contains the start vertex s of p_{i+1} is already in C'. Since the composites in C' are reordered in the same order as in C, we can know s by replacing the containing composite of $ex_{i+1} \in p_{i+1}$, say $c_f \in C$, with its image $c'_f \in C'$ (ln. 20). The path walk given a start vertex s is stated above.

There is an isomorphism if and only if all q paths are found as specified in $P \in \varrho$ (ln. 16). If so, some actions can be performed. In our case, an (m,n)-type hyperedge e that represents the match is created by RECORDMATCH(C', ϱ) (ln. 17), and the hyperedge map $\mathbf{E}$ has to be updated correspondingly. The CPM algorithm continues with the next $t_j \in T$. The match is recorded in $e \in \mathbf{E}$ so that the algorithm only needs to record the most recent match in C'. Before a new search with the next candidate, C' is cleared (ln. 5). The algorithm terminates after it iterates through all the candidates.

[41] Line is abbreviated as ln., lines as lns.

B Miscellaneous Composite Finding Algorithm

Algorithm 2 The MERGE Function of e_{M2} to e_{M1}, two distinct misc composites

```
1  function MERGE(e_M1, v_i, w_j, e_M2)
2      E_1P ← E_1P ∪ E_2P
3      B_1en ← (w_1, ··· , w_{j−1}) ‖ B_1en ‖ (w_{j+1}, ··· , w_{m_2})
4      B_1ex ← (v_1, ··· , v_{i−1}) ‖ B_2ex ‖ (v_{i+1}, ··· , v_{n_1})
5      P_1 ← P_1 ∪ P_2 ∪ (v_i, w_j)
6  end function   ▷ The ‖ symbol denotes the concatenation of two ordered sets.
```

Algorithm 3 The MERGE Function of e_P to e_M, a point composite and a misc composite

```
1  function MERGE(e_M, v_i, w_j, e_P)
2      e'_M ← MERGE(e_P)
3      MERGE(e_M, v_i, w_j, e'_M)
4  end function
5
6  function MERGE(e_P)
7      e_M ← (E_P ← ∅, P ← ∅, B_en ← ∅, B_ex ← ∅)
8      B_en ← entry vertices of e_P
9      B_ex ← exit vertices of e_P
10     for all u ∈ B_en do
11         E_P ← E_P ∪ (u, e_P)
12     end for
13     return e_M
14 end function
```

Algorithm 4 The CLOSECYCLE Function

```
1  function CLOSECYCLE(e_M, v_i, w_j)
2      B_en ← B_en \ w_j
3      B_ex ← B_ex \ v_i
4      P ← P ∪ (v_i, w_j)
5  end function
```

In the Miscellaneous Composite Finding (MCF) Algorithm (Alg. 5), we use a misc composite map $\mathbf{E_M}$ to keep traces of merging (ln. 1). This map contains a set of (hyperedges of) misc composites E_M indexed independently by their constituent point composites $e_P \in e_M \in E_M$, i.e., each entry in E_M has the form (e_P, e_M). At the start of the algorithm, $\mathbf{E_M}$ is empty. E_P is the set of (hyperedges of)

point composites under consideration and $\boldsymbol{b}$ is the bound of path distance[42].

A DFS walk starts with an unvisited $e_P \in E_P$ (lns. 2∼6). The GETNEXTCONNECTEDPOINTENTRY($e_P, \boldsymbol{b}$) function searches for point composites that are connected (by independent paths within $\boldsymbol{b}$, Section 3.3.3) to the exit vertices of e_P. It returns an ordered set of the connected entry vertices EN. If an element[43] in EN_{next} is empty, it simply means that a qualified path is not found for the corresponding exit vertex of e_P. If no path is found, we continue with the next e_P (ln. 2); otherwise, e_P "becomes" a misc composite e_M (ln. 8), since the connected point composite will be later merged into it. The newly created e_M is recorded in $\mathbf{E_M}$ indexed by e_P (ln. 9). We shall walk deeper in the current "depth-first tree" along the path(s) indicated by EN. (We are at the tree root.) We do so by passing on the "subgraph" e_M and the terminal vertices of the corresponding path (ex, en) to the WALKTREE function (ln. 14).

The WALKTREE function (Alg. 6) walks deeper the tree branch until an unqualified path or a backward or a cross tree path (Cormen et al., 2001) is reached. Recall that the function is invoked with a misc composite e_M, an exit vertex v of e_M, and an entry vertex w of a point composite, where (v, w) is a qualified path.

The point composite e_P that cotaines w can be found in the hyperedge map $\mathbf{E}$ (ln. 2). If e_P is in the misc composite map $\mathbf{E_M}$ (ln. 3), then it is visited and is already merged into a misc composite. And if this misc composite is by chance e_M itself (ln. 4), then (v, w) is a backward path and we use CLOSECYCLE to include this path in e_M. When e_P is in another e'_M, we merge e'_M into e_M (ln. 12). Since e'_M is indexed in $\mathbf{E_M}$ by all its point composites $e \in E_P \in e'_M$, we need to replace all indexed values to e_M (lns. 8∼11). When e_P is not in any misc composite, we merge e_P into e_M and record this merge (lns. 15, 16). Now we can further explore the exit vertices of e_P (ln. 17), the similar situation as at a root vertex, but this time e_P is already merged into a misc composite. The walk goes on as explained until all possible branches are visited.

C Model Generation Algorithm

The Model Generation Algorithm (MGA) creates a model map $\mathbf{M}$ (ln. 1) whose entries are added when a `TrackSegment` is generated either during the DFS (ln. 16) or along with the model generation for a hyperedge (ln. 22).

At each DFS walk step, the GENERATEMODELTREE function is passed on with a vertex v in $\mathbf{T}$ and an end node en (ln. 8). The vertex v has a generated model, say V, and it leads the current branch of the tree walk. The end node en is either that of V[44] or an end node of the parent model M of V (i.e., V is in M) to which V is connected with. In both cases, en is the end node that the next generated model shall be coupled with. We may encounter one of the four situations for a walk step (see also Example 2 in 4.1.4):

(1) When v does not have a descendant vertex, a sink is generated (lns. 10∼12).

(2) When v has a descendant vertex t that is not in a hyperedge[45], a track segment is generated and the walk goes one step further (lns. 15∼18).

(3) When v has a descendant vertex t in a hyperedge e, and a model for t is not generated, a model M for e is generated and the walk goes one step further to one of the branches, (lns. 21∼30).

(4) When v has a descendant vertex t in a hyperedge e, and a model for t is generated, there is a back walk (lns. 32∼34).

Note that v can have at most one descendant vertex since all the point composites are in hyperedges where the DFS does not step into. Model generation for hyperedges (ln. 22) is explained in Section 4.1.2. In case of an intersection model, the configuration of the control unit is included in the model generation. The MGA terminates when all the DFS trees rooted from the source vertices are explored.

[42] For example, this distance can be set to 10 or 20 meters.

[43] Note that EN_{next} has at most two elements since the number of exit vertices of a point composite is at most two.

[44] A source or track model can have only one end node.

[45] The only chance for a back walk is when t is in a hyperedge, i.e., a vertex that is not in a hyperedge can not be revisited.

Algorithm 1 The CPM Algorithm

Require: ϱ

1 $C' \leftarrow (c'_1 \leftarrow \emptyset, c'_2 \leftarrow \emptyset, \cdots, c'_w \leftarrow \emptyset)$ ▷ a set of empty elements the same size as $C \in \varrho$
2 ▷ suppose $ex_1 = c_{a,ex_x} \in p_1$, $c_a \in C$, $a \in [1, w]$
3 $T = (t_1, t_2, \cdots, t_r) \leftarrow ex_x$ of all c from $\mathbf{P_F}, \mathbf{P_T}, \mathbf{P_C}$, or $\mathbf{E}$ according to the class of c_a
4 **for** $j = 1 \rightarrow r$ **do** ▷ check all possible candidates $t_j \in T$
5 clear C' ▷ reset C' to empty
6 $s \leftarrow t_j$ ▷ start vertex s of the first search path p_1
7 $c'_a \leftarrow c(s)$ ▷ see ln. 2, cf. ln. 15
8 **for** $i = 1 \rightarrow q$ **do** ▷ check all paths $p_i \in P \in \varrho$
9 $d \leftarrow 0$
10 $t \leftarrow s$
11 **while** t in $\mathbf{T}$ **and** $d < \boldsymbol{b}$ **do** ▷ path search of p_i within bound $\boldsymbol{b} \in \varrho$
12 $t \leftarrow \mathbf{T}(t)$ ▷ $\mathbf{T}$ has an entry $(t, \hat{t})$
13 **if** t in $\mathbf{P_F}, \mathbf{P_T}, \mathbf{P_C}$, or $\mathbf{E}$ **then**
14 **if** $c(t)$ is in the same class as c_b **and** t is en_y of $c(t)$ ▷ see ln. 15
and $c(t) \notin C' \backslash c'_b$ **then** ▷ a match of p_i
15 $c'_b \leftarrow c(t)$ ▷ suppose $en_i = c_{b,en_y} \in p_i$, $c_b \in C$, $b \in [1, w]$
16 **if** $i = q$ **then** ▷ all q paths matched, i.e., a match of pattern ϱ
17 RecordMatch(C', ϱ) ▷ do something related to the match
18 ▷ the loop goes to next t_{j+1} (ln. 4) after ln. 26
19 **else** ▷ suppose $ex_{i+1} = c_{f,ex_x} \in p_{i+1}$, $c_f \in C$
20 $s \leftarrow ex_x$ of $c'_f \in C'$ ▷ start vertex s of the next search path p_{i+1}
21 ▷ the loop goes to p_{i+1} (ln. 8) after ln. 26
22 **end if**
23 **else** ▷ a non match of p_i
24 $i \leftarrow q$ ▷ force the loop to go to next t_{j+1} (ln. 4) after ln. 26
25 **end if**
26 **break**
27 **else**
28 $d \leftarrow d(t) + d$ ▷ accumulate path distance; search continues at ln. 11
29 **end if**
30 **end while**
31 ▷ a non match of p_i because the path ended (with a sink) or bound $\boldsymbol{b}$ is reached
32 $i \leftarrow q$ ▷ force the loop to go to next t_j (ln. 4)
33 **end for**
34 **end for**

- The text following a ▷ symbol is comment.
- A prime sign ($'$) is used next to the original symbols of the elements in the pattern ϱ, to denote a placeholder for the images of the elements (in case of a match); e.g., C' holds images of elements in C (ln. 1).
- The "class of composite c", e.g., ln. 3, refers to whether c is an *fp*, *tp*, *cp* or a hyperedge composite e.
- The "composite c that contains t" is denoted by $c(t)$, e.g., ln. 14.

Algorithm 5 The MCF Algorithm

Require: E_P, $\boldsymbol{b}$

$\mathbf{E_M} \leftarrow \varnothing$ ▷ the misc composite map
for $e_P \in E_P$ **do** ▷ each DFS walk is rooted with an unvisited e_P
  if e_P in $\mathbf{E_M}$ **then**
    continue ▷ go to the next e_P
  end if
  $EN \leftarrow$ GetNextConnectedPointEntry(e_P,$\boldsymbol{b}$) ▷ the entry vertices are ordered
  if EN is not empty **then**
    $e_M \leftarrow$Merge(e_P) ▷ a new e_M is created with e_P, Alg. 3
    $\mathbf{E_M} \leftarrow \mathbf{E_M} \cup (e_P, e_M)$
    for $en \in EN$ **do**
      if $en \neq \varnothing$ **then** ▷ the entry vertex en of a connected point composite
        $i \leftarrow$ index of en in EN ▷ the order of en
        $ex \leftarrow B_{ex,i} \in e_P$ ▷ the corresponding exit vertex ex of e_P
        WalkTree(e_M, ex, en) ▷ Alg. 6
      end if
    end for
  end if
end for

Algorithm 6 The WalkTree Function

function WalkTree(e_M, v, w)
  $e_P \leftarrow$ the indexed value of w in $\mathbf{E}$ ▷ the connected point composite
  if e_P in $\mathbf{E_M}$ **then** ▷ e_P is in a misc composite
    if e_P in e_M **then** ▷ the misc composite is e_M
      CloseCycle(e_M, v, w) ▷ Alg. 4
    else
      $e'_M \leftarrow$ the indexed value of e_P in $\mathbf{E_M}$
      for all $e \in E_P \in e'_M$ **do** ▷ the point composites that are merged into e'_M
        $\mathbf{E_M} \leftarrow \mathbf{E_M} \backslash (e, e'_M)$
        $\mathbf{E_M} \leftarrow \mathbf{E_M} \cup (e, e_M)$ ▷ replace the indexed values
      end for
      Merge(e_M, v, w, e'_M) ▷ Alg. 2
    end if
  else
    Merge(e_M, v, w, e_P) ▷ Alg. 3
    $\mathbf{E_M} \leftarrow \mathbf{E_M} \cup (e_P, e_M)$
    $EN \leftarrow$ GetNextConnectedPointEntry(e_P,$\boldsymbol{b}$) ▷ the same as Alg. 5 ln. 6
    for $en \in EN$ **do** ▷ the same as Alg. 5 lns. 10∼15
      if $en \neq \varnothing$ **then**
        $i \leftarrow$ index of $en \in EN$
        $ex \leftarrow B_{ex,i} \in e_P$
        WalkTree(e_M, ex, en)
      end if
    end for
  end if
end function

Algorithm 7 The MGA Algorithm

Require: **T**, **E**

```
1  M ← ∅                                                          ▷ the model map, see Section 4.1.4
2  for all s do in source vertices in T                           ▷ each DFS walk is rooted with a source vertex s
3      S ← generate a Source instance for s
4      en ← the end node of S
5      GenerateModelTree(s, en)                                   ▷ see ln 8
6  end for
7
8  function GenerateModelTree(v, en)                              ▷ cf., Section 4.1.4
9      t ← T(v)                                                   ▷ check whether T has an entry (v, t)
10     if t = ∅ then                                              ▷ nothing is connected to v
11         SK ← generate a Sink instance
12         couple en and the start node of SK                     ▷ the tree walk ends at this branch
13     else
14         e ← E(t)                                               ▷ check whether E has an entry (t, e)
15         if e = ∅ then                                          ▷ t is not an entry vertex of an hyperedge
16             T ← generate a TrackSegment instance for t
17             couple en and the start node of T
18             GenerateModelTree(t, the end node of T)            ▷ the walk goes to next vertex
19         else
20             T ← M(t)                                           ▷ check whether t (hence e) has a model generated
21             if T = ∅ then                                      ▷ generate a model M for hyperedge e; cf., Section 4.1.2
22                 M ← generate an InfraComponent instance according to the type of e
23                 T ← M(t)                                       ▷ the model T is newly generated along with M
24                 sn ← the start node to which T is connected in M
25                 couple en and sn
26                 for all i = 1 → the size of the end nodes of M do
27                     v ← the i-th exit vertex of e
28                     en ← the i-th end node of M                ▷ assume exit vertices and end nodes are simply ordered with indexes
29                     GenerateModelTree(v, en)                   ▷ the walk goes to one of the branches
30                 end for
31             else                                               ▷ a model for the hyperedge e is already generated
32                 M ← parent model of T
33                 sn ← the start node to which T is connected in M
34                 couple en and sn                               ▷ a back walk: the walk ends at this tree branch
35             end if
36         end if
37     end if
38 end function
```