*Proceedings of the 2015 Winter Simulation Conference*
*L. Yilmaz, W. K. V. Chan, I. Moon, T. M. K. Roeder, C. Macal, and M. D. Rossetti, eds.*

# TOWARDS A UNIFYING FRAMEWORK FOR DEMAND-DRIVEN DIRECTED TRANSPORT (D3T)

Andreas Sorge
Debsankha Manik
Stephan Herminghaus
Marc Timme

Max Planck Institute for Dynamics & Self-Organization
Am Faßberg 17
Göttingen, 37077, GERMANY

## ABSTRACT

Decentralized autonomous systems have the potential to be more performant, resilient and sustainable than classical systems under central control. For example, this includes vehicle routing problems in logistics systems, courier services, and public transport. Here, we outline elements of a framework to model, simulate, and assess the performance of such demand-driven directed transport (D3T) systems. Our contribution is a common language based on the Discrete-Event System Specification, which facilitates modelling and performance analysis of transportation systems across domains. We expect that our approach is useful to identify universal mechanisms and to explore the intricate interplay of the transport system structure at hand and its dynamics.

## 1 INTRODUCTION

Studying and designing transport systems is a multidisciplinary endeavor. The quest to optimize transportation of goods and people in facilities, cities, regions, or whole nations, unites engineers, operation researchers, mathematicians, computer scientists, urban and regional planners, policy-makers and physicists. They all bring their expertise and disciplinary language to the table in the on-going extensive efforts to understand distributed, self-organizing systems. These complex systems under study are themselves as diverse as the disciplines involved, including autonomous production and distribution logistics, urban courier services, and demand-responsive transport in human transit (Hülsmann et al. 2011, Pillac et al. 2013, Santi et al. 2014, Häme 2013, Hyytiä et al. 2010). In general, models of these systems are analytically intractable, and the optimization problems involved are NP-hard and dynamical (Berbeglia et al. 2010), such that the method of choice to analyze these systems is discrete-event simulation. Seemingly, what the field has been lacking so far, is a common high-level language: A language that allows to state the given transportation model and the optimization problem at hand in an accessible but mathematically exact form, and at the same time immediately translates into an executable simulation model. This way, attention focusses on the dynamics, rather than on a purely structural optimization problem description. On the other hand, such a language would add structure to simulation studies hitherto seemingly conducted in an ad-hoc fashion.

This Paper is a contribution towards such a language and implementation. In the following, we outline elements of a modelling framework for demand-driven directed transport systems. We first characterize the phenomenological transport dynamics and introduce the constituent elements of our framework (Section 2). Building upon this specification, we utilize and build upon the general P-DEVS language to arrive at a formal specification of our framework (Section 3).

## 2 THE DEMAND-DRIVEN DIRECTED TRANSPORT (D3T) FRAMEWORK

### 2.1 Overview

The Demand-Driven Directed Transport framework is a framework for modelling transportation systems that serve transport requests of discrete immotile loads in a physical transport space. In the system, transporters process requests by travelling along paths of subsequent origins and destinations and transporting the loads.

In particular, demand-driven directed transport (*D3T*), has the following properties:

- Transportation is *demand-driven*: there are no fixed services
- Transported objects are *discrete loads*: there is no continuous quantity such as water or electricity,
- The transported loads are immotile: Loads do not move on their own,
- Loads are transported by *discrete transporter units*: there is no conveyer belt, or pipes (such as the Internet distributes data packets on a continuous basis).
- Transportation is *directed*: there is no change of transporters.

This framework provides a unifying mathematical language to model D3T systems and assess their performance. Formally, the D3T framework is a subset of the mighty, general *Parallel Discrete Event-System Specification (P-DEVS)* language (Chow and Zeigler 1994). (In the following, when we speak of DEVS models, we refer to P-DEVS models.) A crucial feature of the D3T framework is its strict *modularity*: it specifies the abstract component classes and the protocol for interactions of component modules. A D3T modeller may either implement her own component modules, or choose from a set of pre-defined (and programmed) modules to specify a D3T simulation.

In fact, we aim to provide a solid technical foundation for D3T modelling and simulation such that no further modelling and programming is needed to build a whole D3T model and run a full-scale D3T simulation.

### 2.2 D3T Phenomenological Transport Dynamics

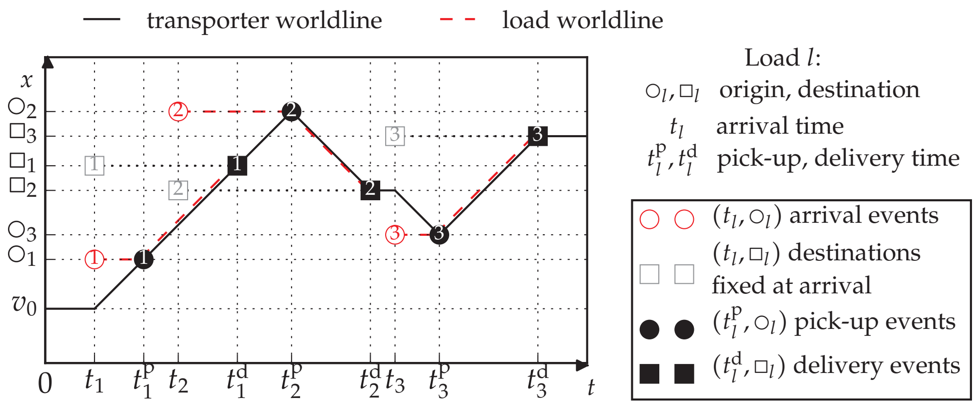
transporter worldline
load worldline
x
$\bigcirc_2$
$\square_3$
$\square_1$
$\square_2$
$\bigcirc_3$
$\bigcirc_1$
$v_0$
0
$t_1$
$t_1^p$
$t_2$
$t_1^d$
$t_2^p$
$t_2^d$
$t_3$
$t_3^p$
$t_3^d$
t
Load $l$:
$\bigcirc_l, \square_l$ origin, destination
$t_l$ arrival time
$t_l^p, t_l^d$ pick-up, delivery time
$(t_l, \bigcirc_l)$ arrival events
$(t_l, \square_l)$ destinations fixed at arrival
$(t_l^p, \bigcirc_l)$ pick-up events
$(t_l^d, \square_l)$ delivery events

Figure 1: A trajectory of a simple D3T model with only 1 transporter and 3 transport requests $(l, \bigcirc_l, \square_l)$. The transporter is a myopic taxi that processes transport requests on the real line $\mathbb{R}$ on a first-come-first-served basis, one at a time. If all known requests have been served, the transporter remains idle at its current position. The taxi also needs to travel empty to pick-up a new load $l$ at its origin $\bigcirc_l$, and deliver it at its destination $\square_l$. The Figure shows origin and destination at the arrival time $t_l$. The point $(t_l^p, \bullet)$ in space-time marks the pick-up event at the load origin, and the point $(t_l^d, \blacksquare)$ in space-time marks the delivery event at the load destination.

In D3T systems, vehicles transport discrete loads (or passengers) in the transport space from their respective origin to their respective destination. These loads arrive to the system according to a stochastic birth process, with their origin and destination determined and fixed at arrival time (see Figure 1 for a simple example). A D3T model is a *dynamic* model in the sense of a dynamic optimization problem: not all information is available, but only reveils in the course of the evolution of the system: a load arrival is only known at the arrival time, and not before.

Load arrivals are independent of load transport. While load arrivals are stochastic, load transport is deterministic. Load arrivals are the only external events influencing the transportation dynamics: the load arrival process feeds into the deterministic transportation dynamics.

A transporter picks up a load at its origin, and delivers a load at its destination. Once picked up, loads remain on-board the transporter until it reaches the destination of the load. Loads do not change transporters.

Which of the transporters picks up a given load, is subject to the control policy. The D3T framework allows the control policy to reject load requests. If rejected, a load leaves the transport system immediately. The travel time between any two points of the transport space is the metric distance. In geodesic geometries, transporters travel at unit speed along geodesic segments, i.e. shortest paths between subsequent waypoints. In networks, transporters jump along the nodes of the shortest path, where the jump duration is the arc weight.

Even though loads arrive according to a continuous-time stochastic process, and transporters move continuously in geodesic geometries, it is discrete-time events governing the transportation dynamics. Transporters pick-up and deliver loads, depart and arrive at positions, at intrinsically discrete instants of time.

## 2.3 The Transport Space

Demand-driven directed transport (D3T) takes place in a physical space. The mathematical model of such a *transport space* is a hemimetric space $\mathcal{M}$ with a hemimetric $d$ (Deza and Deza 2013). In particular, a hemimetric is a metric that does not need to be neither discernible nor symmetric. Additionally, the transport space $\mathcal{M}$ shall be either "continuous" (geodesic), or discrete (a network).

A *geodesic hemimetric space* $\mathcal{M}$ is a hemimetric space for which any two distinct points $x, y$ are connected by a *geodesic directed segment* (or shortest path) from $x$ to $y$. For example, the Euclidian space $\mathbb{R}^n$ with the standard metric $d(x,y) = \sqrt{\sum_{i=1}^{n}(y_i - x_i)^2}$ is a geodesic metric space for all $n \in \mathbb{N}$.

While geodesic metric spaces allow for continuous movements along geodesic segments, networks are discrete metric spaces which only allow jumps along *paths* of links (*arcs*) between their elements (*nodes* or *vertices*). A *network* $G = (V, E, \omega)$ is a weighted digraph which is simple and strongly connected. The network is *undirected*, if $\forall e = (v, v') \in E : e' = (v', v) \in E, \omega(e') = \omega(e)$. A network $G = (V, E, \omega)$ is endowed with the hemimetric of the shortest-path length, yielding a hemimetric space.

## 2.4 Transport Requests and Loads

Each transport request $(l, \bigcirc_l, \square_l)$ defines a point-like material entity: a *load*. A load $l$ is immotile and demands active transport from its *origin* $\bigcirc_l \in \mathcal{M}$ to its *destination* $\square_l \in \mathcal{M}$ in the transport space $\mathcal{M}$. At time of arrival, a load requests transport immediately, The *load index* $l \in \mathbb{N}$ enumerates the transport requests in order of arrival. Transport requests arrive to a D3T system according to a stochastic process, and precisely, according to a marked point process $\{(T_n, Y_n), n \in \mathbb{N}\}$ (Jacobsen 2006). The random variable $T_n$ is the arrival epoch of the $n$-th load. If $T_n = \infty$ for some $n$, no more loads arrive. The random variable $Y_n$ denotes the *mark* $(\bigcirc_n, \square_n, \sigma_n)$ of the $n$-th load, from the augmented *mark space* $\mathcal{M} \times \mathcal{M} \times \Sigma \cup \{\nabla\}$. The elements of $\Sigma$ are additional marks that the control policy and/or performance analysis may take into account. For example, a D3T model may specify a mark for priority customers. Note that $Y_n = \nabla$ if and only if $T_n = \infty$.

## 2.5 Transporters

**General properties** A transporter is a point-like motile object in the transport space $\mathcal{M}$. Its purpose is to actively transport immotile loads. Each transporter $i$ is endowed with an integer (or infinite) *capacity* $C_i$, which is the maximum number of loads it can transport at the same time. A D3T model has fixed number $N$ of transporters which might differ in their capacity but are otherwise identical.

In the D3T framework, transporters are actively transporting the loads, but nevertheless they are myopic passive agents that merely execute the control policy. A transporter typically lacks any but short-term knowledge about its own queue of loads to transport and positions to travel to. Furthermore, transporters do not interact with other objects in the transport space, i.e. loads or other transporters. Transporters are also ideal in the sense that they are always in service, and they do not need to refuel or the like, unless explicitly told to do so. A transporter either moves at unit velocity, or remains at its current position $v \in \mathcal{M}$. In geodesic spaces, transporters move continuously along geodesic segments. In networks, transporters perform discrete jumps along nodes of shortest paths, where each jump takes a time given by the weight of the respective arc.

**Jobs** This framework describes transporter operation as a sequence of jobs. Each job $\mathrm{j} = (\mathbb{P}, \tilde{v}, \mathbb{D})$ is a tuple of

1. a *pick-up set* $\mathbb{P}$ of loads to pick up at the current location $v$ of the transporter,
2. the scheduled destination $\tilde{v}$ to travel to,
3. a *delivery set* $\mathbb{D}$ of loads to deliver at the new location $\tilde{v}$.

At any time $t$, a transporter either has a current transporter job to process, or it is idle. The set $\tilde{\mathbb{P}}$ denotes the *loads scheduled for pick-up* at the current position $v$, the position $\tilde{v} \in \mathcal{M}$ the destination to travel to, and the set $\tilde{\mathbb{D}}$ denotes the *loads scheduled for delivery* at the destination $\tilde{v}$.

**Picking up and delivering loads** Picking up and delivering loads takes the transporter a certain amount of time. The pick-up period and delivery period may depend on the position $v \in \mathcal{M}$ and on the set of loads to pick-up ($\mathbb{P}$) or deliver ($\mathbb{D}$). The period does not depend on the particular transporter. The *pick-up period function* (*delivery period function*) $\tau_P(v, \mathbb{P})$ ($\tau_D(v, \mathbb{D})$) determines the time it takes a transporter to pick up (deliver) loads $\mathbb{P}$ ($\mathbb{D}$) at the position $v$.

**Queue** Each transporter has a queue $Q$ of transporter jobs to process. Formally, at any time $t$ the queue $Q = \left(\tilde{j}_n\right)_n$ is a finite sequence of transporter jobs $\tilde{j}_n = \left(\tilde{\mathbb{P}}, \tilde{v}, \tilde{\mathbb{D}}\right)$ scheduled for the transporter to process sequentially in ascending order. The job that the transporter currently processes is not part of the queue (any more).

**Dispatcher action** As stated before, the transporters merely execute the control policy. In practice, they receive commands from the dispatching unit (called *dispatcher*). In the D3T framework, there are the following dispatcher actions a transporter willingly accepts at any time:

1. appending a sequence of jobs $J$ to the transporter queue $Q \mapsto (Q, J)$ (*submit*),
2. replacing the queue $Q \mapsto J$ with a new sequence of jobs $J$ (*submit and replace*)
3. emptying the queue $Q \mapsto \varnothing$ (*submit and replace empty job sequence*),
4. canceling the current job $c \mapsto 1$ (*cancel*). This implies also emptying the queue ($Q \mapsto \varnothing$) and makes the most sense if the dispatcher submits a new sequence of jobs at the same time.

Note that the first 3 actions only modify the queue, but do not affect the current job. The last action is the most intrusive, regarding the ongoing operation of the transporter.

**Transporter states and state transitions** At any time $t$, the state of a transporter is given by the variables in Table 1. A transporter experiences the following state transitions (see Table 2 and Figure 2): A transporter starts a job ($*$), when it has an non-empty queue ($Q \neq \varnothing$) and either has just ended a job (†) or has been idle (I). Starting a job, a transporter transits to the pick-up phase (P) when the set of loads

Table 1: Transporter state.

| Symbol | State variable |
|---|---|
| $v \in \mathcal{M}$ | Current position |
| $w \in \mathbb{R}_0$ | Waiting time (remaining jump time) to arrive at $\tilde{v}$ |
| $\mathbb{L}$ | Current cargo (set of loads on-board the transporter) |
| $Q$ | Current queue of transporter jobs |
| $\tilde{\mathbb{P}}$ | Current job: Set of loads scheduled for pick-up at current position |
| $\tilde{v} \in \mathcal{M}$ | Current job: Destination |
| $\tilde{\mathbb{D}}$ | Current job: Set of loads scheduled for delivery at destination |
| $M$ | Current *mode*, see Table 2. |
| $c \in \{0,1\}$ | A Boolean variable that indicates whether the dispatcher ordered the transporter to cancel its current job. |

Table 2: Transporter modes. We introduce the pseudo-modes for the start of processing the current job ($*$) and end of processing the current job (†). The *lifetime* of a mode is the period a transporter spends in that mode before it advances to the next mode. The *idle* mode is only left at external input from the dispatcher, when it adds jobs to the transporter queue. At the end of a mode, the transporter state autonomously changes according to the *effects* column.

| Symbol | Mode | Lifetime | Entry condition | Effect |
|---|---|---|---|---|
| **I** | idle | $\infty$ | $Q = \varnothing \wedge w = 0 \wedge \tilde{v} = v \wedge \tilde{\mathbb{P}} = \tilde{\mathbb{D}} = \varnothing$ | |
| $*$ | (job start) | 0 | $Q \neq \varnothing$ | $(\tilde{\mathbb{P}}, \tilde{v}, \tilde{\mathbb{D}}) \mapsto \tilde{j}_1, Q \mapsto (\tilde{j}_n)_{n=2}^{\lvert Q \rvert}$ |
| **P** | pick-up | $\tau_P(v, \tilde{\mathbb{P}})$ | $\tilde{\mathbb{P}} \neq \varnothing \wedge \neg c$ | $\mathbb{L} \mapsto \mathbb{L} \cup \tilde{\mathbb{P}}, \tilde{\mathbb{P}} \mapsto \varnothing$ |
| **T** | travel | $d(v, \tilde{v})$ | $\tilde{\mathbb{P}} = \varnothing \wedge \tilde{v} \neq v \wedge \neg c$ | $v \mapsto \tilde{v}$ |
| **D** | delivery | $\tau_D(\tilde{v}, \tilde{\mathbb{D}})$ | $\tilde{\mathbb{P}} = \varnothing \wedge \tilde{v} = v \wedge w = 0 \wedge \tilde{\mathbb{D}} \neq \varnothing \wedge \neg c$ | $\mathbb{L} \mapsto \mathbb{L} \setminus \tilde{\mathbb{D}}, \tilde{\mathbb{D}} \mapsto \varnothing$ |
| † | (job end) | 0 | $(\tilde{\mathbb{P}} = \varnothing \wedge \tilde{v} = v \wedge w = 0 \wedge \tilde{\mathbb{D}} = \varnothing) \vee c$ | $c \mapsto 0$ |

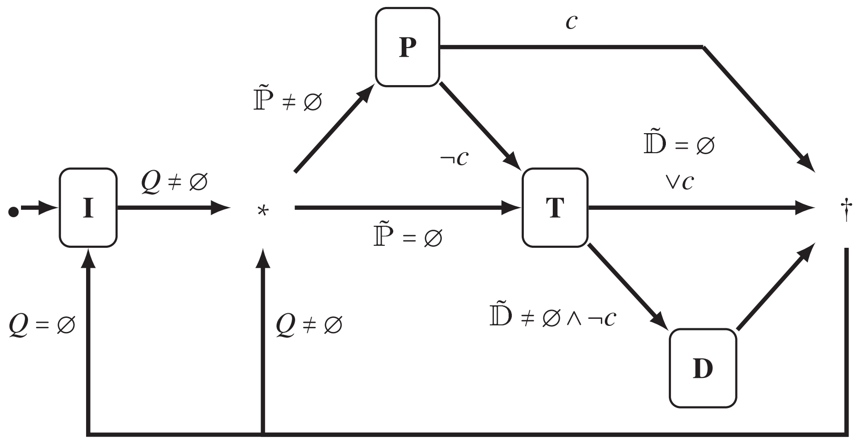
P
c
$\tilde{\mathbb{P}} \neq \varnothing$
$\neg c$
$\tilde{\mathbb{D}} = \varnothing$
$\vee c$
I
$Q \neq \varnothing$
$*$
T
†
$\tilde{\mathbb{P}} = \varnothing$
$Q = \varnothing$
$Q \neq \varnothing$
$\tilde{\mathbb{D}} \neq \varnothing \wedge \neg c$
D

Figure 2: Transition diagram of a transporter, see Table 1 for the state variables and Table 2 for the modes. Labels at the transition arcs are the entry condition for the successive mode.

scheduled for pick-up is non-empty ($\tilde{\mathbb{P}} \neq \varnothing$). After pick-up, the transporter starts moving (T), unless it has been ordered to cancel ($c$). If there are not any loads to pick up ($\tilde{\mathbb{P}} = \varnothing$), the transporter directly transits to the travel phase (T). After travelling, the transporter transits to the delivery phase (D), unless there are not any loads to deliver ($\tilde{\mathbb{D}} = \varnothing$), or it has been ordered to cancel ($c = 1$). A transporter ends a job (†) after delivery, unless there have not been any loads scheduled for delivery, or the transporter was ordered to cancel. If the queue is empty ($Q = \varnothing$) after ending a job, the transporter becomes idle.

When the dispatcher orders to cancel a job, the transporter queue is emptied. Furthermore,

1. if the transporter is picking up loads (**P**), it regularly finishes pick-up and, prematurely, ends the current job thereafter;
2. if the transporter is travelling (**T**), it stops as soon as possible (in a geodesic geometry this means immediately, in a network this means when the current jump to the next node is completed);
3. if the transporter is delivering loads (**D**), it regularly finishes delivery and ends the current job thereafter.

## 2.6 Control Policy

The *control policy* $\mathcal{D}$ is a set of rules that determine the instructions for the transporters given the load arrivals up to the current time. The control policy covers each load arrival: it either rejects a load request, or assigns the load to a transporter (at least eventually). It is the control policy that governs the transporter queues. In the light of new load arrivals, the control policy may modify transporter queues, or even cancel the job currently processed by any transporter.

In a simulation model, the *dispatcher* is the component that embodies the control policy. In a D3T model, the dispatcher contains all the *model-specific logic* of the D3T transport. Therefore, its internal state is typically rather intricate and highly model-specific. For sophisticated control policies, the dispatcher will even contain an own concept of the world: in this framework, the dispatcher does not "own" the transporters, their states are hidden from the dispatcher. The only way to know about the transporter states is via signalling.

## 2.7 Summary: Parameters of a D3T Model

To conclude this Section, a D3T model $\mathrm{M} = \left((\mathcal{M}, d), \Sigma, (\mathcal{T}, \mathcal{Y}), N, \{C_i\}_{i=1}^{N}, \left\{v_i^0\right\}_{i=1}^{N}, \tau_P, \tau_D, \mathrm{D}_{\mathcal{D}}, \mathcal{O}\right) \in \hat{\mathrm{M}}$ is specified by the following parameters:

- the transport space $\mathcal{M}$ (either geodesic or network) and the associated hemimetric $d$,
- the set $\Sigma$ of additional load marks and the load arrival process $(\mathcal{T}, \mathcal{Y}) = \{(T_n, Y_n), n \in \mathbb{N}\}$ on the mark space $\mathcal{M} \times \mathcal{M} \times \Sigma$,
- the number $N$ of transporters,
- the capacities $\{C_i\}_{i=1}^{N}$ of the transporters with $C_i \in \mathbb{N} \cup \{\infty\}$,
- the initial positions $\left\{v_i^0\right\}_{i=1}^{N}$ of the transporters with $v_i^0 \in \mathcal{M}$,
- the pick-up period function $\tau_P$,
- the delivery period function $\tau_D$,
- the control policy $\mathcal{D}$ (the dispatcher model $\mathrm{D}_{\mathcal{D}}$),
- and additionally, we also include the set of observer models $\mathcal{O} = \left\{\mathrm{O}_j\right\}_j$ that record simulation data for performance analysis.

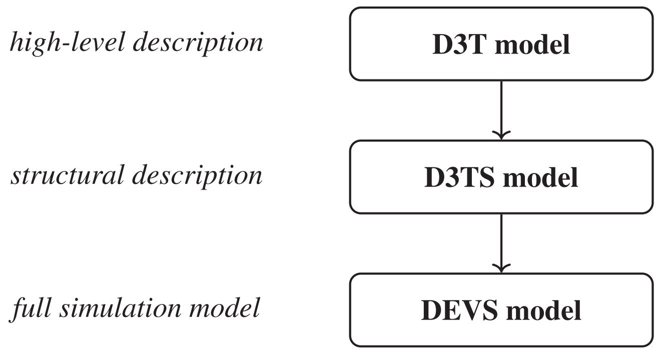

Figure 3: Three-tier description of a D3T model, which maps unto a D3TS model as described in this Section, and ultimately, unto a low-level but fully-fledged DEVS simulation model.

## 3 THE D3T SYSTEM SPECIFICATION (D3TS)

### 3.1 Overview

A D3TS model is a formal mapping of a D3T model onto a system of interacting components in the system-theoretic sense (Wainer 2009, Zeigler, Praehofer, and Kim 2000). The DEVS description of a D3TS model is the complete description for simulation of a D3T model (cf. Figure 3). In other words, our contribution here is a common formal language (D3TS model) for specifying demand-driven directed transport models in the general formal P-DEVS language. The existing P-DEVS framework in turn comes with an abstract simulation algorithm and general implementations such as the *adevs* C++ library (Nutaro 2010, Nutaro, J. J. 2013).

The D3TS specification defines generic *component classes* of specific D3T model components. There are four pairwise disjoint *component classes*:

- Load source class $\hat{\mathrm{L}}$, containing the *load sources*, the simulation components which embody the load arrival process;
- Dispatcher class $\hat{\mathrm{D}}$,
- Transporter class $\hat{\mathrm{T}}$,
- Observer class $\hat{\mathrm{O}}$.

One of the characteristic design traits of DEVS and the D3T Specification is that each simulation component is modelled as a black box: Internal states are inaccessible, the only way of knowing about the system and its other components is via input and output events. This specification provides several *event types*. Examples of events are load arrivals (*requests*) or a transporter delivering loads (*delivery*).

Given an event type e, all event instances $e \in \mathrm{e}$ of that type originate from component instances C of only one class $\hat{\mathrm{C}}_{\mathrm{e}}$. For example, only load source instances output request events, and only transporter instances output delivery events. The component classes and their event types define the D3TS framework (Figure 4).

### 3.2 Event Types

This Section introduces all event types e of this framework (Table 3).

**Request event** A load source instance L outputs an event instance $e_?$ of the *request* event type ? whenever a load arrives. The mark is $(\sigma_l, \bigcirc_l, \square_l)$. The output event instance is $e_? = (l, \sigma, \bigcirc, \square)$ with the load index $l$, the load origin $\bigcirc = \bigcirc_l$ and the load destination $\square = \square_l$.

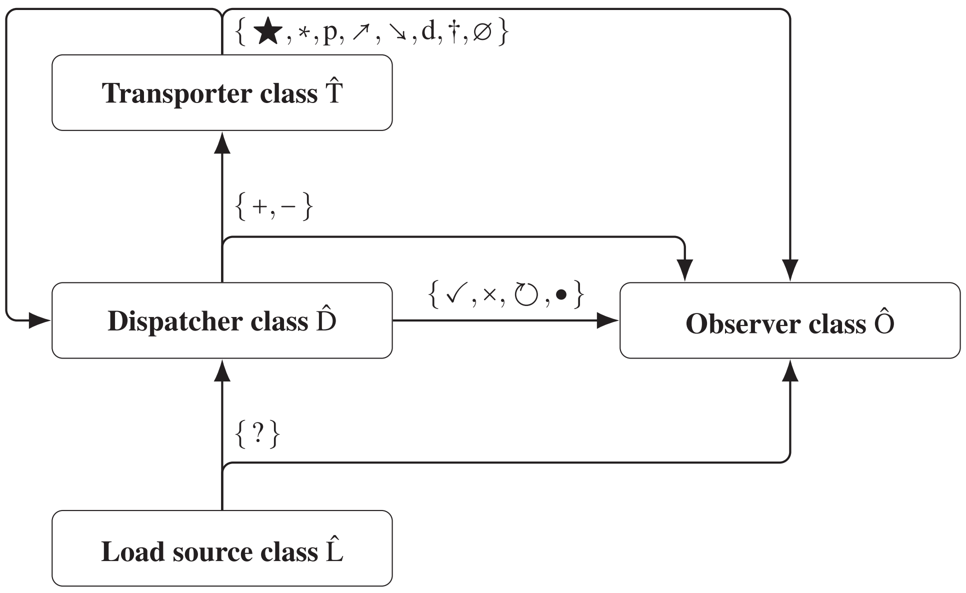

Figure 4: The D3T system framework.

**Assign event** A dispatcher instance D outputs an event instance $e_{\checkmark} = (l, i)$ of the *assign* event type $\checkmark$ whenever it assigns a load $l$ for transport by transporter $i$. This event confirms the load request, i.e. assignment implies acceptance of the load request.

**Reject event** A dispatcher instance D outputs an event instance $e_{\times} = (l)$ of the *reject* event type $\times$ whenever it rejects a load request $l$.

**Submit event** A dispatcher instance D outputs an event instance $e_{+} = \left(i, \tilde{\mathrm{j}}, r\right)$ of the *submit* event type $+$ to submit a sequence of jobs $\tilde{\mathrm{j}}$ to a transporter $i$. If the boolean *replace* variable $r$ is *true*, the submitted jobs replace the transporter queue $Q_i$. Otherwise, the submitted jobs are appended to the queue.

**Cancel event** A dispatcher instance D outputs an event instance $e_{-} = (i)$ of the *cancel* event type $+$ to order a transporter $i$ to cancel processing its current job. Hence, this command is particularly useful in combination with a submit event that replaces the current transporter queue. In this combination, the transporter receives new instructions to follow immediately.

**Busy event** A dispatcher instance D outputs an event instance $e_{\circlearrowright} = (i)$ of the *busy* event type $\circlearrowright$ to report that a transporter $i$ has become busy.

**Idle event** A dispatcher instance D outputs an event instance $e_{\bullet} = (i)$ of the *idle* event type $\bullet$ to report that a transporter $i$ has become idle.

**Init event** At the begin of the simulation, a transporter instance $\mathrm{T}_i$ outputs an event instance $e_{\bigstar} = (i, v, C)$ of the *init* event type $\bigstar$ to report initialization of transporter $i$ at position $v$ with capacity $C$. Together with the init event instance, the transporter instance outputs an empty queue event instance (see below).

**Job start event** A transporter instance $\mathrm{T}_i$ outputs an event instance $e_{*} = (i, j)$ of the *job start* event type $*$ when transporter $i$ starts processing job no. $j$.

**Pick-up event** A transporter instance $\mathrm{T}_i$ outputs an event instance $e_{\mathrm{p}} = (i, j, \mathbb{P})$ of the *pick-up* event type p when transporter $i$ finishes picking up the loads $\tilde{\mathbb{P}}$ of its current job $j$.

**Departure event** A transporter instance $\mathrm{T}_i$ outputs an event instance $e_{\nearrow} = (i, j, \tilde{v}, \tilde{\tau})$ of the *departure* event type $\nearrow$ when transporter $i$ departs from its current position $v$ to travel to the destination $\tilde{v}$ of its current job $j$. The expected travel time is $\tilde{\tau} = d(v, \tilde{v})$.

**Arrival event** A transporter instance $\mathrm{T}_i$ outputs an event instance $e_{\searrow} = (i, j, v)$ of the *arrival* event type $\searrow$ whenever transporter $i$ arrives (and stops) at a position $v$. The transporter either stops at its scheduled destination $\tilde{v}$, or when the dispatcher canceled its current job $j$.

Table 3: Event types and their instance data tuples, names, and producing component classes.

| Name | e | $\hat{\mathrm{C}}_{\mathrm{e}}$ | Instance $e \in \mathrm{e}$ |
|---|---|---|---|
| request | ? | $\hat{\mathrm{L}}$ | $(l, \sigma, \circ, \square)$ |
| assign | ✓ | $\hat{\mathrm{D}}$ | $(l, i)$ |
| reject | $\times$ | $\hat{\mathrm{D}}$ | $(l)$ |
| submit | $+$ | $\hat{\mathrm{D}}$ | $(i, \tilde{\mathrm{j}}, r)$ |
| cancel | $-$ | $\hat{\mathrm{D}}$ | $(i)$ |
| busy | $\circlearrowleft$ | $\hat{\mathrm{D}}$ | $(i)$ |
| idle | $\bullet$ | $\hat{\mathrm{D}}$ | $(i)$ |
| init | $\bigstar$ | $\hat{\mathrm{T}}$ | $(i, v, C)$ |
| job start | $\ast$ | $\hat{\mathrm{T}}$ | $(i, j)$ |
| pick-up | p | $\hat{\mathrm{T}}$ | $(i, j, \mathbb{P})$ |
| departure | $\nearrow$ | $\hat{\mathrm{T}}$ | $(i, j, \tilde{v}, \tilde{\tau})$ |
| arrival | $\searrow$ | $\hat{\mathrm{T}}$ | $(i, j, v)$ |
| delivery | d | $\hat{\mathrm{T}}$ | $(i, j, \mathbb{D})$ |
| job end | † | $\hat{\mathrm{T}}$ | $(i, j)$ |
| empty queue | $\varnothing$ | $\hat{\mathrm{T}}$ | $(i)$ |

**Delivery event** A transporter instance $\mathrm{T}_i$ outputs an event instance $e_{\mathrm{d}} = (i, j, \mathbb{D})$ of the *delivery* event type d whenever transporter $i$ finishes delivering the loads $\tilde{\mathbb{D}}$ of its current job $j$.

**Job end event** A transporter instance $\mathrm{T}_i$ outputs an event instance $e_{\dagger} = (i, j)$ of the *job end* event type † when transporter $i$ ends processing the current job $j$. This also includes jobs that have been prematurely canceled by the dispatcher instance.

**Empty queue event** A transporter instance $\mathrm{T}_i$ outputs an event instance $e_{\varnothing} = (i)$ of the *empty queue* event type $\varnothing$ whenever transporter $i$ has ended processing its current job and finds its queue empty, $Q_i = \varnothing$. It also outputs an empty queue event instance at time of initialization (as transporter instances are initialized with empty queues).

**Transporter events** We refer to the event types $\{\bigstar, \ast, \mathrm{p}, \nearrow, \searrow, \mathrm{d}, \dagger, \varnothing\}$ as *transporter events*.

## 3.3 Mapping a D3T Model to a D3T System

A D3T system contains a certain number of instances of the underlying D3TS component model classes, as specified by mapping the D3T model. Note that the behavior of a component model $\mathrm{C}(\mathrm{M})$ for a given D3T model M depends on the other component models and the other parameters of the D3T model.

**Load sources** Given a D3T model M, the load source model $\mathrm{L}(\mathrm{M})$ embodies the marked point process $\{(T_n, Y_n), n \in \mathbb{N}\}$. A D3T system can contain multiple load sources at the discretion of the modeler. Load sources do not have any input events.

**Dispatcher** A dispatcher component model $\mathrm{D}_{\mathcal{D}} \in \hat{\mathrm{D}}$ implements a specific control policy $\mathcal{D}$. In each D3T system, there is only one instance $\mathrm{D}_{\mathcal{D}}(\mathrm{M})$ of the dispatcher model. The framework requires each dispatcher model to output submit, assign, busy, and idle events. The dispatcher model may further output cancel and/or reject events, depending on the control policy.

**Transporter** This framework specifies exactly one transporter component model $\mathrm{T} \in \hat{\mathrm{T}}$. For a given D3T model M, there is a transporter instance $\mathrm{T}_i \in \mathrm{T}(\mathrm{M})$ for each of the $N$ transporters. An instance $\mathrm{T}_i$ is initialized with initial position $v_i^0$ and empty cargo $\mathbb{L} = \varnothing$ at time $t = 0$.

**Observer** An observer listens to simulation events but does not interfere with the simulation: it is the simulator's tool to record simulation statistics, instead of accessing simulation components' states directly. This design pattern facilitates and enforces encapsulation and the black-box paradigm. There are various observer component models $\mathrm{O} \in \hat{\mathrm{O}}$. Each D3T model M may specify multiple observer models $\mathrm{O}_i \in \hat{\mathrm{O}}$, and

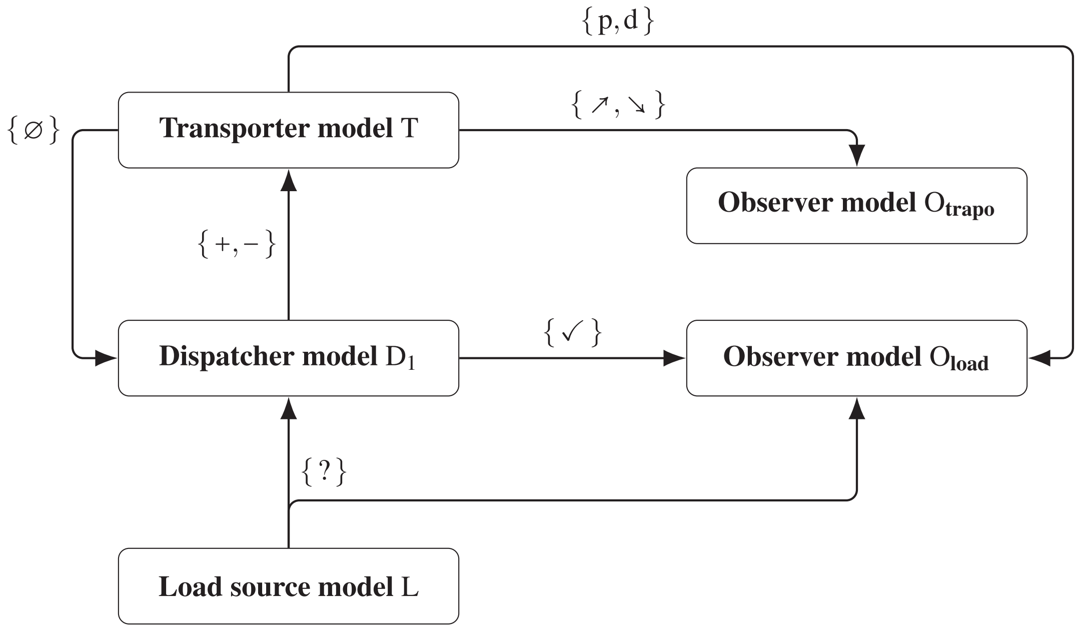

Figure 5: D3TS model graph of the example D3T model with the component models as nodes and arcs labelled by the event types.

in the D3T system there will be exactly one instance $O_i \in O_i(M)$ of each observer model. As observers are passive, they do not affect the dynamics of the D3T system. For a D3T model, the observer models merely specify which observables are accessible for recording.

## 3.4 A D3T Model Example

Let us illustrate the technicalities of the D3TS component and event framework with a concrete example $M_1$ of a D3T model. As there is only one load source model $L \in \hat{L}$ and one transporter model $T \in \hat{T}$, we only need to specify the dispatcher model $D_1 \in \hat{D}$ and the observer models $\mathcal{O}$. Specifically, let the number of transporters be $N = 3$. Let the dispatcher model be $D = D_1$, and let the set of observer models be $\mathcal{O} = \left\{ O_{load}, O_{trapo} \right\}$.

Let us consider a control policy which cancels transporter jobs (− event type), and reacts upon transporters reporting empty queues (∅ event type). Let observer model $O_{load}$ record load statistics of arrivals, assignments, pick-ups and deliveries, and let observer model $O_{trapo}$ record transporter statistics such as departures and arrivals. These component input/output ports determine the *D3TS model graph* (Figure 5) which is a directed graph without loops.

## 3.5 On the Mapping of a D3TS Model to a DEVS Model

To complete the presentation of our framework, we sketch the technical mapping of a D3T model instance M to a parallel DEVS network model $N(M) = (X_N, Y_N, D, \mathcal{I}, Z)$ with ports. Each port $p$ in the DEVS network model is an event type. The DEVS network model coupling functions treat all event instances independently and only couple output ports to input ports of the same event type. The DEVS network model itself neither takes input nor produces output ($X_N(M) = Y_N(M) \equiv \varnothing$.) The set of components $D(M)$ of the network model contains a component $d$ for each component instance C in the D3T system as specified in Section 3.3. The family of sets of influencers $\mathcal{I}$ directly derives from the D3T system framework (Figure 4). A coupling function $z_{d,d'} \in Z$ transforms output of the DEVS network component $d$ to input of the DEVS network component $d'$. Within the component and event framework it is again particulary simple to define this transformation. From any output bag $y^b$ that $d$ generates, it selects all events $(p,v)$ whose type $p$ is implemented as input at the receiving end.

The load source component model generates the load arrival events. In fact, every instance of the DEVS network model (i.e. every simulation run) implements one realization of the load arrival process, through the load source. Note that it is in this sense that the deterministic DEVS formalism accounts for random events: the DEVS model instance is formally initialized with a single realization of the stochastic process. However, the software implementation typically generates the respective random events on the fly. Within the constraints of this framework regarding the event types, the D3T modeler is free to specify dispatcher and observer component models as generic DEVS atomic models. Finally, the transporter DEVS model derives from the transporter states (Table 1), including modes (Table 2), and the transition graph (Figure 2).

This completes the outline of the steps necessary to arrive at a formal DEVS description of a D3T system.

## 4 CONCLUSION AND FUTURE WORK

In this Paper, we outlined our approach to a unifying language to model demand-driven directed transport systems (Section 2) across disciplines and domains. We further introduced elements of a formal description of such systems based on the parallel discrete-event system specification (Section 3). What we did not include here is a fully-fledged formal mapping of such a D3TS model to the underlying P-DEVS model, as this is straightforward and becomes rather technical. We also omitted actual control policies and performance measures. A comparative analysis of various control policies implemented in the D3T framework is work in progress. Furthermore, we envision a modular simulation program, which accepts the parameters of the transport system such as number of vehicles, the topology, or the stochastic arrival process, and the algorithmic control policy as input and provides a unified API to the practitioner. A prototype implementation in Python (*pyd3t*) based on a Python port of Jim Nutaro's excellent C++ implementation of DEVS is being developed in our group. Such further extensions notwithstanding, we expect the present Paper to provide the sound theoretical foundation for a unifying framework for demand-driven directed transport, useful to modelling and simulation practitioners across disciplines.

## ACKNOWLEDGMENTS

AS and DM acknowledge current and former members of the D3T team: F. Fix, J. A. Kassel, B. Walter, and M. Wendland. MT and AS acknowledge the support of the Organization for Research on Complex Adaptive Systems (or-cas).

## REFERENCES

Berbeglia, G., J.-F. Cordeau, and G. Laporte. 2010. "Dynamic pickup and delivery problems". *European Journal of Operational Research* 202 (1): 8–15.

Chow, A. C., and B. P. Zeigler. 1994. "Parallel DEVS: a Parallel, Hierarchical, Modular Modeling Formalism". In *Proceedings of the 1994 Winter Simulation Conference*, edited by J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila, 716–722. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Deza, M. M., and E. Deza. 2013. *Encyclopedia of Distances*. Berlin, Heidelberg: Springer.

Häme, L. 2013. *Demand-Responsive Transport: Models & Algorithms*. Ph. D. thesis, School of Science, Aalto University, Espoo, Finland.

Hülsmann, M., B. Scholz-Reiter, and K. Windt. (Eds.) 2011. *Autonomous Cooperation and Control in Logistics*. Berlin, Heidelberg: Springer.

Hyytiä, E., A. Penttinen, and R. Sulonen. 2010. "Congestive Collapse and Its Avoidance in a Dynamic Dial-a-Ride System with Time Windows". In *Analytical and Stochastic Modeling Techniques and*

*Applications*, edited by K. Al-Begain, D. Fiems, and W. Knottenbelt, Volume 6148 of *Lecture Notes in Computer Science*, 397–408. Berlin, Heidelberg: Springer.

Jacobsen, M. 2006. *Point Process Theory and Applications*. Boston, Massachusetts: Birkhäuser.

Nutaro, J. J. 2010. *Building Software for Simulation*. Hoboken, New Jersey: Wiley.

Nutaro, J. J. 2013. "adevs (A Discrete Event Simulator) Library". Accessed Jul. 15, 2015. http://web.ornl.gov/%7e1qn/adevs/.

Pillac, V., M. Gendreau, C. Guéret, and A. L. Medaglia. 2013. "A Review of Dynamic Vehicle Routing Problems". *European Journal of Operational Research* 225 (1): 1–11.

Santi, P., G. Resta, M. Szell, S. Sobolevsky, S. H. Strogatz, and C. Ratti. 2014. "Quantifying the Benefits of Vehicle Pooling with Shareability Networks". *Proceedings of the National Academy of Sciences* 111 (37): 13290–13294.

Wainer, G. A. 2009. *Discrete-Event Modeling and Simulation*. Boca Raton, Florida: CRC Press.

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of Modeling and Simulation*. Second ed. Amsterdam: Academic Press.

## AUTHOR BIOGRAPHIES

**ANDREAS SORGE** is a Ph.D. candidate in Physics at the Max Planck Institute of Dynamics & Self-Organization and Georg-August-Universität Göttingen, Germany. He holds a Bachelor's degree in Mathematical Physics from University of Canterbury, New Zealand, and a Diplom in Physics from Georg-August-Universität Göttingen. He is the current Chair of the Organization for Research on Complex Adaptive Systems (or-cas). His research interests include sustainable transport, self-organization in complex systems and networks. He combines methods from statistical physics, queueing theory and discrete-event simulation, and enjoys developing free scientific software packages to advance reproducible computational science. His e-mail address is as@ds.mpg.de.

**DEBSANKHA MANIK** is a Ph.D. candidate in Physics at the Max Planck Institute of Dynamics & Self-Organization and Georg-August-Universität Göttingen, Germany. He holds a MS degree in Physics from IISER Kolkata, India. His research interests include dynamics of complex flow networks, emergence of synchrony in nonlinear dynamical systems and transport in biological systems. His e-mail address is dmanik@nld.ds.mpg.de.

**STEPHAN HERMINGHAUS** is Head of the Department for Dynamics of Complex Fluids at the Max Planck Institute for Dynamics & Self-Organization and an Adjunct Professor of Physics at the Georg-August-Universität Göttingen, Germany. His research interests include microscopic processes and mechanisms of self-organization in physical and biological systems. His e-mail address is stephan.herminghaus@ds.mpg.de.

**MARC TIMME** is Head of the Max Planck Research Group for Network Dynamics at the Max Planck Institute for Dynamics & Self-Organization and an Adjunct Professor of Physics at Georg-August-Universität Göttingen. Currently, he is Vice Chair of the Physics of Socio-Economics Systems Division of the German Physical Society (DPG) and the Organization for Research on Complex Adaptive Systems (or-cas). His research interests include the theory of network dynamics and principles of self-organization, which he applies to power grids, transport systems, social networks, biological and artificial neural networks. His e-mail address is timme@nld.ds.mpg.de.