Consistency Management Framework for Database Used in Integrated Simulations

Itsuki Noda

Information Technology Research Institute
National Institute of Advanced Industrial Science and Technology
2-41-6 Aomi, Koto-ku, Tokyo 135-0064, Japan
i.noda@aist.go.jp

Abstract. In this article, I propose a mathematical framework of consistency management that guarantees validity among data that are used in integrated simulation systems. When we apply integrated simulations to real-time prediction/evaluation of complex social phenomena like disaster and rescue, checking and keeping consistency of data is an important issue to validate simulation results, because multiple and delayed information are reported to the database continuously in real-time applications. The proposed formalization gives fundamental background of consistency and validity of database and simulation. I also investigate about cost of the management in two major implementation styles.

1 Introduction

In an integrated social simulation like RoboCupRescue Simulation, data management is one of important issues, because various heterogeneous sub-simulators interact with each other. In a Japanese national special project for earthquake disaster mitigation in urban areas (DDT project), we are developing an integrated disaster-and-rescue simulation system [1, 2], in which a number of simulators for various phenomena, for example, damages of ground and road, TSUNAMI, fire-ignition, liquefaction, and so on, are used to provide initial data for the multi-agent rescue simulation. In the case where we apply such an integrated simulation to real-time prediction/evaluation of real phenomena, we must pay attention to management of consistency of dataset used in sub-simulators in order to guarantee the validity of total simulations.

The consistency of data-set can be broken by the following features of the application:

- Several data can be reported for a single phenomena. Some of the data may include noise or may be wrong.
- Some data reach to rescue headquarters with large delay. Such data may conflict with assumed setting of simulations that run before the arrival of the data.

In such case, each session of the simulation is grounded to the different dataset. This means that we need to handle carefully the simulation result.

A. Bredenfeld et al. (Eds.): RoboCup 2005, LNAI 4020, pp. 279–289, 2006.

Even in the case where we use simulations off-line, we face similar situation when we run many sessions using various initial datasets. In an integrated simulation, some sub-simulations use results of other simulations. Therefore, we need to bring together results of sub-simulations for another sub-simulation carefully not to use inconsistent datasets.

In this article, I will give a formalization of *version control* of datasets to keep consistency among datasets. In the rest of this paper, requirements for the version control are figured out by listing several use-cases of integrated simulations in section 2. Section 3 shows definitions and theories about *version* of datasets and its control. Costs of several operation of version control are investigated in section 4, and discussed from various point of view in section 5.

2 Requirements

In order to figure out the concept and requirement of *version control*, I will point out some use-cases of datasets in a database that is used with a integrated simulation.

2.1 Use-Case 1: Noise in Sensing

In real-time applications, data are not always true. In general, different sensors report different data for the same phenomenon. Sometimes, the difference among these data is significantly large, so that we can not handle these data as a distribution of the same phenomenon. In such a case, we must consider some data are true and others are false. Generally, it is difficult to distinguish true/false of each data.

For example, 5 and 7 were reported alternately as a value of a seismic intensity of a certain aftershock at a certain point in Chuetsu Earthquake in Nov. 2004. In such case, results of simulations based on intensity 5 and 7 are quite different. Generally, we must choose one of 5 or 7 as a true value of the intensity because it is generally meaningless to use an average of 5 and 7.

Choice of values causes another issue on the integrated simulation. Suppose that there are phenomena X, Y, Z, W. In an integrated simulation, Y and Z can be estimated from X value by simulations independently, and W can be calculated from Y and Z. Suppose X is sensed by two sensors a and b, which report different values x_a and x_b, respectively. Here, a simulation calculates Y using x_a and outputs y_a as its value. In the same time, another simulation calculates Z as z_b based on x_b. In this case, we should avoid calculating value W by y_a and z_b, because these values are grounded on different values of phenomenon X. For example, we should not integrate results of two simulations, estimated damage of road based on intensity 5 and estimated number of victims under intensity 7.

2.2 Use-Case 2: Delay of Report

Because sensors are located far from the database, the sensed values are reported with delay. Especially in applications to disaster and rescue, the delay may be

very large. Actually, damages of Yamakoshi village in Niigata were reported to head quarters of rescue two days after Chuetsu Earthquake.

In such cases, not all data are available for a certain simulation, so that we must estimate lacked data by interpolating or other methods to run the simulation. When the estimated data turns out to be different from the real data that are reported with delay, we must ignore results of the simulation and re-calculate using the new real data.

2.3 Use-Case 3: Complex Dependency

In an integrated simulation, we may have deep dependencies among sub-simulations. For example, in DDT project[1, 2], the simulation system consists of more than 10 sub-simulations, which are connected with each other via a database.

In such case, relation of dependency described in section 2.1 becomes more complex. In addition, large-scaled simulations like social ones handle huge dataset. Therefore, checking mechanism of consistency should be scalable and light-weight.

2.4 Summary of Requirements

Based on the above discussions about the use-cases, we can summarize that management systems of dataset should have:

- a facility to realize a kind of version control to manage choices of dataset,
- a way to check consistency among simulations and dataset,
- a facility to extract dataset that is consistent with an existing dataset, and
- a mechanism to keep consistency of existing dataset when a delayed sensor data is inserted into the database.

3 Formalization

In this section, I give a formalization of version control of dataset, and derive theorems about consistency, both of which enable to handle requirements examined in the previous section.

3.1 Database and Snapshot

We suppose that a *database* is a collection of *data*, each of which indicates a sensed or simulated value about a *thing* at a certain time. We also suppose that the *database* is dynamic. In other words, sensors and simulators put new *data* into the database continuously [1].

Formally, these words are defined as follows:

[1] We assume that data in a database are never updated or deleted. When various values for a certain event are reported, then all values are stored independently in the database.

Thing: A thing θ is an identifier of an object, feature, or phenomena. For example, "building X", "speed of car Y" and "seismic intensity at a point Z" are *things*. A thing can consist of other things according to a structure of a corresponding object or phenomena.

Event: An event e is a snapshot of a thing θ at a time t. An event e can be notated as a tuple $\langle \theta, t \rangle$. For example, "damage of human X at time t" "speed of car Y at time t", and "the degree of blockade of road Z at time t" are *events*.

Data: A data $d_{\theta,t,s}$ is a value of event $\langle \theta, t \rangle$ acquired by a sensor or simulation s. Because a sensed or simulated value is affected by several conditions and noises, multiple *data* for a certain *event* can exist. When the data comes from a sensor, s indicates an identifier of the sensor. When the data is estimated by a simulation, s is a *version* defined in the next section.

When we indicate a time to store the data into a database, we use the following notation:

$$d_{\theta,t,s;\tau},$$

where τ indicates the time when the data is stored into a database.

Database: A database $\mathbf{DB} = \{d_{\theta,t,s;\tau}\}$ is defined as a set of whole data stored in the database.

$\mathbf{DB}_{(0,\mathcal{T})} = \{d_{\theta,t,s;\tau} \in \mathbf{DB} | \tau < \mathcal{T}\}$ indicates a snapshot of the database at a certain time $\mathcal{T}$. In other words, $\mathbf{DB}_{(0,\mathcal{T})}$ is a set of data that are stored into $\mathbf{DB}$ before $\mathcal{T}$.

3.2 Version

We say a session of a simulation to indicate a run of the simulation using certain initial data and settings. Generally, a session is performed as follows:

1. Collect a set of data $\boldsymbol{D}_{\text{ground}}$ about some events $\boldsymbol{E}_{\text{ground}} = \{e_i | i\}$ from $\mathbf{DB}$ at the begin time of the simulation ($\mathcal{T}_{\text{begin}}$).
2. Estimate a set of data $\boldsymbol{D}_{\text{target}}$ about other events $\boldsymbol{E}_{\text{target}} = \{e_i | i\}$.
3. Store $\boldsymbol{D}_{\text{target}}$ into $\mathbf{DB}$ at the end time of the simulation ($\mathcal{T}_{\text{end}}$).

We call these set of data of input or output of a session of a simulation as a *version*. In other words, a session receives a version from a database as a ground of simulation and inserts another version of data into the database as a result of the simulation.

When we conduct an integrated simulation that consists of several sessions of sub-simulations, we need to pay attention to guaranteeing that each session is grounded on a right dataset. For example, a sub-simulation may depends on a collection of results of other sub-simulations as shown in section 2. In this case, we need to avoid that some results are grounded on inconsistent data.

In order to make it easy to check such consistency among dataset, we define *consistency* among *versions* as follows:

Version: A version v is defined as a tuple as follows:

$$\langle \boldsymbol{E}, \mathcal{T}, \boldsymbol{D}, \boldsymbol{G} \rangle,$$

where $\boldsymbol{E} = \{e_i|i\}$ is a set of events, $\boldsymbol{D}$ is a dataset under the version, $\mathcal{T}$ is a time when **DB** is accessed to manipulate the dataset, and $\boldsymbol{G} = \{u_i|i\}$ is a set of versions on which $\boldsymbol{D}$ is directly grounded. These elements must satisfy the following condition:

$$\boldsymbol{D} \subset \mathbf{DB}_{(0,\mathcal{T})}/E = \{d_{\theta,t,s;\tau} |\, \langle \theta, t \rangle \in \boldsymbol{E}, \tau < \mathcal{T}\}$$

Generating a New Version: There are four types of operation to generate a new version, *sprout*, *extraction*, *production*, and *union*.

A version is *sprouted* when a new dataset is collected from **DB**, where all data in the dataset should be *primary one that come from sensors directly*. A sprouted version consists of a tuple:

$$\langle \boldsymbol{E}, \mathcal{T}, \boldsymbol{D}, \phi \rangle \,,$$

where $\boldsymbol{E}$ is a set of related events, $\boldsymbol{D}$ is a collected dataset from **DB**, and $\mathcal{T}$ is a time to collect $\boldsymbol{D}$.

A version v can be *extracted* from another version $u = \langle \boldsymbol{E}_u, \mathcal{T}_u, \boldsymbol{D}_u, \boldsymbol{G}_u \rangle$, where the dataset of v ($\boldsymbol{D}_v$) is a subset of Ds_u. The extracted version v is denoted as a tuple:

$$\langle \boldsymbol{E}_v, \mathcal{T}_v, \boldsymbol{D}_v, \{u\} \rangle \,,$$

where $\boldsymbol{D}_v \subseteq \boldsymbol{D}_u$, $\boldsymbol{E}_v = \{e|d_{\theta,t,s} \in \boldsymbol{D}_v, e = \langle \theta, t \rangle\}$, and $\mathcal{T}_v$ ($> \mathcal{T}_u$) is a time of extraction.

When a session of a simulation outputs a version v as a result using version u as a ground, we call the version v as a *production* of the version u, or denote $u \rhd v$. The *production* v is defined as a tuple:

$$u \rhd v \Leftrightarrow v = \langle \boldsymbol{E}_v, \mathcal{T}_v, \boldsymbol{D}_v, \{u\} \rangle \,,$$

where $\boldsymbol{E}_v$ is a set of simulated events, $\boldsymbol{D}_v$ is the result of the simulation, and $\mathcal{T}$ is a time to finish the simulation [2].

A *union* of two versions u and v ($u \oplus v$) is a version to imply a sum of the two versions. A *union* is a virtual version that includes empty sets of events and data, so that the unified version is constructed as follows:

$$u \oplus v = \langle \phi, \mathcal{T}, \phi, \{u, v\} \rangle \,,$$

where $\mathcal{T}$ is a time to unify two versions.

We define a version v is *grounded* in version u (or denote $u \succ v$) iff v is generated by *union*, *production*, or *extraction* from u. As defined above, the following relation is satisfied:

$$u \succ v \Leftrightarrow u \in \boldsymbol{G}_v$$

where $v = \langle \boldsymbol{E}_v, \mathcal{T}_v, \boldsymbol{D}_v, \boldsymbol{G}_v \rangle$.

[2] We suppose that results of simulations are stored immediately after the simulations.

A *footmark* is a relation of two versions which are linked recursively by groundness relationships as follows: We say that version u is a *footmark* of version v iff

$$u \precnsim^* v \Leftrightarrow \begin{cases} u = v \\ \text{or} \\ u \succ u', u' \precnsim^* v \end{cases}$$

We use a *trail* as a set of a version ($\{u_i|i\} = \mathbf{Tr}(v)$) iff when each version u_i in the set is a footmark of version v.

Consistency: We define that two versions, u and v, are *primarily consistent* (or denote $u \dot{\sim} v$) iff

$$\forall e \in \boldsymbol{E}_u \cap \boldsymbol{E}_v : \boldsymbol{D}_u/e = \boldsymbol{D}_v/e,$$

where $u = \langle \boldsymbol{E}_u, \mathcal{T}_u, \boldsymbol{D}_u, \boldsymbol{G}_u \rangle$ and $v = \langle \boldsymbol{E}_v, \mathcal{T}_v, \boldsymbol{D}_v, \boldsymbol{G}_v \rangle$.

We define that two versions, u and v, are *consistent* (or denote $u \sim v$) iff

$$\forall u' \precnsim^* u, \forall v' \precnsim^* v : u' \dot{\sim} v'.$$

In order to guarantee a unified version $u \oplus v$ is dependable, u and v should be consistent.

We define that a version v is *self-consistent* iff

$$v \sim v.$$

3.3 World Line

As mentioned at the definition of *data*, a certain event can have multiple *data* in a **DB**. Some of these data are treated as true and others are ignored in analyses and simulations. We use the word '*world line*' to indicate a set of data that are treated as true and taken into account for the analysis.

Because multiple analyses can be conducted in parallel, there can exist multiple *world lines* on a **DB**. While these *world lines* are inconsistent with each other, all data in a *world line* should be consistent. Here, '*consistent*' means that

> *each data in a world line is sensed or simulated based on the same conditions and dataset.*

The purpose of the version control is to check and keep the consistency when a simulation extends a world line.

Formally, *world line* and related concepts are defined as follow:

World Line: A world line $\boldsymbol{w}$ is any subset of a database **DB**, that is, $\boldsymbol{w} \in \mathbf{DB}$. When a world line $\boldsymbol{w}$ includes multiple data for a certain event e, the value of the event e is treated as a distribution of probability by Monte Carlo interpretation. When a world line $\boldsymbol{w}$ includes no data for a certain event e, the event is treated as "don't-care".

There are no restriction for a world line to select data in a database. Therefore, whole set of world line is same as power set of whole data, $2^{\mathbf{DB}}$.

A world line is independent from changes of the database.

Projection of World Line: $\boldsymbol{w}/t = \{d_{\theta,t',s;\tau} \in \boldsymbol{w} | t' = t\}$ indicates a projection of a world line w at time t (or simply, *time slice* of a world at time t).

$\boldsymbol{w}/\theta = \{d_{\theta',t,s;\tau} \in \boldsymbol{w} | \theta' = \theta\}$ indicates a projection of a world line w about a certain thing θ.

$\boldsymbol{w}/e = \{d_{e',s;\tau} \in \boldsymbol{w} | e' = e\}$ indicates a projection of a world line w about a certain event e.

Consistency of World Line. We define a world line $\boldsymbol{w}$ *supports* a version v, or denote $\boldsymbol{w} \vdash v$ iff the following condition is satisfied:

$$\forall u = \langle \boldsymbol{E}_u, \mathcal{T}_u, \boldsymbol{D}_u \rangle \succ^* v, \forall e \in \boldsymbol{E}_u : \boldsymbol{w}/e = \boldsymbol{D}_u/e$$

We define a world line $\boldsymbol{w}$ is *consistent* iff the following condition is satisfied:

$$\forall d_{\theta,t,s;\tau} \in \boldsymbol{w} : s \text{ is a sensor id or } \boldsymbol{w} \vdash s.$$

In other words, each simulated data in a consistent world line should be included by a version supported by the world line.

It is important that only consistent world lines are meaningful in a database. If a world line is inconsistent, that is, some data are not supported by the world line, the world line is useless because the data is not well-grounded into the world line. Such situation should be avoided when we conduct simulations and update database.

3.4 Theorems About Keeping Consistency

Here, I will derive some theorems that show how generated versions are supported by consistent world lines.

Lemma 1. *When a version v is self-consistent and a world line $\boldsymbol{w}$ consists of data that belong to versions in a trail of v, any footmarks of v is supported by $\boldsymbol{w}$.*

$$\begin{aligned} \forall v \ : \ & v \text{ is self-consistent}, \\ & \boldsymbol{w} = \{d | u = \langle \boldsymbol{E}_u, \mathcal{T}_u, \boldsymbol{D}_u, \boldsymbol{G}_u \rangle \succ^* v, d \in \boldsymbol{D}_u\} \\ \rightarrow \ & \forall u \succ^* v : \boldsymbol{w} \vdash u \end{aligned}$$

Proof (Lemma 1). Suppose that there exists a footmark u of the version v, which is not supported by the world line $\boldsymbol{w}$. In other words,

$$\exists u = \{\boldsymbol{E}_u, \mathcal{T}_u, \boldsymbol{D}_u, \boldsymbol{G}_u\} \succ^* v, \exists e \in \boldsymbol{E} : \boldsymbol{w}/e \neq \boldsymbol{D}_u/e.$$

Because of the definition, $\boldsymbol{w}$ is a super set of $\boldsymbol{D}_u$. So, $\boldsymbol{w}/e \neq \boldsymbol{D}_u/e$ is satisfied iff $\exists d \in \boldsymbol{w}/e, d \notin \boldsymbol{D}_u/e$. This means:

$$\exists u' \{\boldsymbol{E}_{u'}, \mathcal{T}_{u'}, \boldsymbol{D}_{u'}, \boldsymbol{G}_{u'}\} \in \mathbf{Tr}(v) : u' \neq u, \\ d \in \boldsymbol{D}_{u'}$$

However, this violates the definition of self-consistency of version v. Therefore, any footmark u is supported by $\boldsymbol{w}$.

Using this lemma, we can derive the following theorem.

Theorem 1. *When a version v is self-consistent, there exists a world line $\boldsymbol{w}$ that is consistent and supports v.*

$$\forall v : v \textit{ is self-consistent} \rightarrow \exists \boldsymbol{w} : \boldsymbol{w} \textit{ is consistent}, \boldsymbol{w} \vdash v$$

Proof (Theorem 1). Consider a world line $\boldsymbol{w} = \{d | u = \langle \boldsymbol{E}_u, \mathcal{T}_u, \boldsymbol{D}_u, \boldsymbol{G}_u \rangle \nleftarrow v, d \in \boldsymbol{D}_u\}$. $\boldsymbol{w}$ supports any footmark of version v because of Lemma 1. On the other hand, for any simulated data $d_{\theta,t,s;\tau} \in \boldsymbol{w}$, the version s to which the data belongs is always in the trail of version v. Therefore, the version s is supported by the world line w.

This theorem tells that we need pay attention only to keeping self-consistency of newly generated versions in order to make the versions are meaningful.

Based on the first theorem, we can derive the following theorems about generation of new versions.

Theorem 2. *Any sprouted version is self-consistent.*

Proof (Theorem 2). Because of all data in a sprouted version come from sensor directly, the version is grounded itself [3].

Theorem 3. *When a version v is extracted from a self-consistent version u, v is self-consistent iff the following condition is satisfied:*

$$\begin{aligned} &\forall v : u \supset v, \\ &\quad v \textit{ is self-consistent} \leftrightarrow \forall e \in \boldsymbol{E}_v : \boldsymbol{D}_u/e = \boldsymbol{D}_v/e, \end{aligned}$$

where u and v are $\langle \boldsymbol{E}_u, \mathcal{T}_u, \boldsymbol{D}_u, \boldsymbol{G}_u \rangle$ and $\langle \boldsymbol{E}_v, \mathcal{T}_v, \boldsymbol{D}_v, \boldsymbol{G}_v \rangle$, respectively.

Proof (Theorem 3). If the condition is satisfied, u and v is consistent. Because u is self-consistent, v is also consistent.

Suppose that the condition is not satisfied. In this case, there exists an event e that has different dataset in u an v, that is $\boldsymbol{D}_u/e \neq \boldsymbol{D}_v/e$. Because $\boldsymbol{E}_v$ is a subset of VE_u, e is included both in $\boldsymbol{E}_v$ and $\boldsymbol{E}_u$. Therefore, u and v is not consistent, so that v is not self-consistent.

Theorem 4. *When a version u is self-consistent, a production v of the version u is self-consistent and there exists a consistent world line that supports both versions v and u.*

$$\begin{aligned} &\forall u : \textit{self-consistent} \rightarrow \\ &\qquad \forall v : u \rhd v \rightarrow v \textit{ is self-consistent}, \\ &\qquad\qquad \exists \boldsymbol{w} : \boldsymbol{w} \textit{ is consistent}, \boldsymbol{w} \vdash v \end{aligned}$$

Here, we suppose that a simulation never estimate data about the same event as one that it takes as input.

[3] Because of this reason, the sprouted version is defined with no grounding versions.

Proof (Theorem 4). Because of the definition, $\mathbf{Tr}(v)$ is $\{v\} + \mathbf{Tr}(u)$. Because $\boldsymbol{E}_u \cap \boldsymbol{E}_v = \phi$, any footmark of version u is consistent with version v. Therefore, the version v is self-consistent. As the result, there exists a consistent world line that supports v because of Theorem 1.

Theorem 5. *When two versions, v and u, are self-consistent and consistent with each other, a union of these versions is self-consistent, and there exists a consistent world line that supports the union.*

$$\forall u, v : \textit{self-consistent}, u \sim v \rightarrow$$
$$u \oplus v \textit{ is self-consistent},$$
$$\exists \boldsymbol{w} : \boldsymbol{w} \textit{ is consistent}, \boldsymbol{w} \vdash u \oplus v$$

Proof (Theorem 5). Because of the definition, $\mathbf{Tr}(u \oplus v)$ is $\{u \oplus v\} + \mathbf{Tr}(u) + \mathbf{Tr}(v)$. Suppose that x and y are versions in $\mathbf{Tr}(u \oplus v)$. When both of x and y are in $\mathbf{Tr}(u)$ or $\mathbf{Tr}(v)$, they are primary consistent with each other because u and v are self-consistent. When one of them is in $\mathbf{Tr}(u)$ and another in $\mathbf{Tr}(v)$, they are primary consistent with each other because u and v are consistent with each other. When one of x and y is identical with $u \oplus v$, they are primary consistent because the version $u \oplus v$ contain no events.

Therefore, the union $u \oplus v$ is self-consistent. As the result, there exists a consistent world line that supports $u \oplus v$ because of Theorem 1.

4 Operation Cost of Version Control

In order to realize version control for integrated simulations, we need to implement several additional functions into the database as follows:

- to store information about version. As its definition, the information should includes a set of related events($\boldsymbol{E}$), access time to the database($\mathcal{T}$), a set of data ($\boldsymbol{D}$), and a set of ground version ($\boldsymbol{G}$). In these items, handling of $\boldsymbol{D}$ is most important because the number of elements in $\boldsymbol{D}$ is generally large.
- to insert new data to the database with keeping self-consistency of each existing version. Especially, data that are reported with large delay should be managed carefully, because events of the data may already be used in some simulation sessions.
- to check consistency of two version. As shown in theorems in the previous section, the critical operation to generate a new version is mainly in *union* operation. To keep self-consistency of unified version, we must check consistency of two versions.

In the followings subsections, we investigate average cost for each operation under two typical implementations of the version control.

4.1 Positive Listing

A strait-forward way to store information about $\boldsymbol{D}$ of a version is to store a list of data (or identifiers of data) in $\boldsymbol{D}$. We call this method as *positive listing*.

Cost to Store Information. The order of the size of storage to keep a list $\boldsymbol{D}$ in the positive listing is $O(|\boldsymbol{D}|)$. This cost become a problem when a simulation handle a large-scale phenomena like earthquake, because the number of data is large in such simulations.

Cost to Insert New Data. When a new data is inserted to **DB**, there occur no additional operations for managing consistency of versions, because the new data does not affect on $\boldsymbol{D}$ of existing version.

Cost to Check Consistency. In order to check consistency between versions u and v, the following operation is required:

$$\forall u' \in \mathbf{Tr}(u), \forall v' \in \mathbf{Tr}(v), \forall e \in (\boldsymbol{E}_{u'} \cap \boldsymbol{E}_{v'}) : \\ \text{check } \boldsymbol{D}_{u'}/e = \boldsymbol{D}_{v'}/e$$

where $u' = \langle \boldsymbol{E}_{u'}, \mathcal{T}_{u'}, \boldsymbol{D}_{u'}, \boldsymbol{G}_{u'} \rangle$ and $v' = \langle \boldsymbol{E}_{v'}, \mathcal{T}_{v'}, \boldsymbol{D}_{v'}, \boldsymbol{G}_{v'} \rangle$. The order of the cost of this operation is $O(\sum |\boldsymbol{D}_{u'}| + \sum |\boldsymbol{D}_{v'}|)$.

4.2 Negative Listing

Another strategy to store information about $\boldsymbol{D}$ is to store a list of data that are related with events in $\boldsymbol{E}$ but do not appear in $\boldsymbol{D}$. We call this method as *negative listing*. In other words, the *negative listing* stores data that are not used in the version.

Generally, such not-used data will occur under the following situation:

- When multiple various data about a certain event is reported by different sensors, some of them may be considered as false data that are not used in a certain simulation session.
- When a simulation starts at time $\mathcal{T}$ using data of a certain event e, other data about e may be reported after $\mathcal{T}$.

In the negative listing method, a list of the following dataset is stored to keep information about version v:

$$\bar{\boldsymbol{D}}_v = \sum_{e \in \boldsymbol{E}_v} (\mathbf{DB}_{(0,\mathcal{T}_v)}/e - \boldsymbol{D}_v)$$

Cost to Store Information. The order of the size of storage to keep a list $\bar{\boldsymbol{D}}$ is $O(|\bar{\boldsymbol{D}}|)$. Generally, it is expected that this cost is smaller than the cost in the positive listing, because the data in $\bar{\boldsymbol{D}}$ is generally exceptional one.

Cost to Insert New Data. When a new data is reported to **DB** after a timestamp $\mathcal{T}_v$ of an existing version v, we may need to handle the new data is in the negative list of v, because the data is not used in the version. However, we can distinguish such negative data by comparing timestamps of the data (τ) and the version ($\mathcal{T}$). Therefore, there occurs no additional operation to manage the consistency.

Cost to Check Consistency. In order to check consistency between versions u and v, the following operation is required:

$$\forall u' \in \mathbf{Tr}(u), \forall v' \in \mathbf{Tr}(v), \forall e \in (\boldsymbol{E}_{u'} \cap \boldsymbol{E}_{v'}) :$$
$$\begin{cases} \text{check } \bar{\boldsymbol{D}}_{v'}/e - \bar{\boldsymbol{D}}_{u'}/e = \mathbf{DB}_{(\mathcal{T}'_u, \mathcal{T}'_v)}/e & \text{if } \mathcal{T}'_u < \mathcal{T}'_v \\ \text{check } \bar{\boldsymbol{D}}_{u'}/e - \bar{\boldsymbol{D}}_{v'}/e = \mathbf{DB}_{(\mathcal{T}'_v, \mathcal{T}'_u)}/e & \text{if } \mathcal{T}'_v < \mathcal{T}'_u \end{cases}$$

where $u' = \langle \boldsymbol{E}_{u'}, \mathcal{T}_{u'}, \boldsymbol{D}_{u'}, \boldsymbol{G}_{u'} \rangle$, $v' = \langle \boldsymbol{E}_{v'}, \mathcal{T}_{v'}, \boldsymbol{D}_{v'}, \boldsymbol{G}_{v'} \rangle$, and $\mathbf{DB}_{(\mathcal{T}_x, \mathcal{T}_y)}$ means a set of data that is inserted between time $\mathcal{T}_x$ and $\mathcal{T}_y$. The order of the cost of this operation is $O(\sum \left|\mathbf{DB}_{(\mathcal{T}_{v'}, \mathcal{T}_{u'})}/\boldsymbol{E}\right| + \sum \left|\bar{\boldsymbol{D}}_{u'}\right| + \sum \left|\bar{\boldsymbol{D}}_{v'}\right|)$.

5 Discussion

It is not simple to determine which of positive and negative listings is better than another. In the case of large scale applications, however, the size of data is generally huge. Therefore, the negative listing has an advantage in both costs of storage and consistency checking.

In the evaluation of costs in section 4, the cost to projection operations (for example, $\mathbf{DB}_{(\mathcal{T}'_u, \mathcal{T}'_v)}/e$) is ignored. This is because general database systems like RDB have sophisticated low-cost facilities using indexing/hashing technique.

The cost to manage a set of events $\boldsymbol{E}$ is also ignored in the above evaluation. While datasets can be handled only by listing, a set of events can be represented a projection axis to the database. For example, in geographical application like disaster-rescue simulation, a set of events can be represented by type of features and area of interest. Therefore, we can assume the operation of the event sets is also abstracted and low-cost on storage and consistency checking.

There are several open issues on the formalization as follows:

- How to realize a facility to extract version that is consistent with a certain version. Such operation will happen when a simulation requires parts of result of two versions.
- How to formalize statistic operations of results of multiple simulations. One of purposes of the simulation is to calculate statistic value like averages and variances based on multiple simulation using different random seeds. Current formalization can not handle it, because such operations break consistency among versions.

References

1. Tadokoro, S.: An overview of japan national special project daidaitoku (ddt) for earthquake disaster mitigation in urban areas. In: Proc. of IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA2003) Workshop on Advanced Robots and Information Systems for Disaster Response. (2003)
2. Tadokoro, S.: Japan government special project: development of advanced robots and information systems for disaster response (daidaitoku) — for earthquake disaster mitigation in urban areas —. In: Proc. of IEEE Workshop on Safety Security and Rescue Robotics (SSRR03). (2003)