Specification of the CSIRO Common Modelling Protocol

AD Moore, DP Holzworth, NI Herrmann,
NI Huth, BA Keating & MJ Robertson

Version: **9 Mar 2005**

1. INTRODUCTION......1
1.1. Formal definition of a simulation......2
1.2. Roles involved in defining and using simulations......3
2. DEFINITION OF ENTITIES WITHIN THE PROTOCOL......4
2.1 Components......4
2.2 Properties......4
2.3 Events and event handlers......5
2.4 Systems and simulations......6
2.5 Messages......6
3. TASKS PERFORMED USING THE PROTOCOL......7
3.1. Initialisation of a simulation......8
3.2. Termination of a simulation......10
3.3. Computation of a time step......11
3.4. Transmission of driving property values......12
3.5. Changing another component's owned property......14
3.6. Transmission of an event......15
3.7. Transmission of an error message......16
3.8. Recording and restoring the model state......17
3.8.1. Recording the model state......17
3.8.2. Restoring the model state......18
Restoring a checkpoint......18
3.9. Changing the model structure......19
3.9.1. Registration of a property or event......19
3.9.2. Deregistration of a property or event......20
3.9.3. Adding a component to a system......21
3.9.4. Removing a component from a system......22
3.9.5. Deactivating a component......23
3.9.6. Activating a component......23
3.10 Obtaining information about properties, components or events......24
4. PROTOCOL MESSAGES......25
4.1. Summary of protocol messages......25
4.2. Protocol messages in detail......26
5. STANDARD PROPERTIES AND EVENTS......38
5.1. Standard component properties......38
5.2. Standard event handlers......38
5.3. Time property......38
6. DEFINITION OF DATA TYPES......40
6.1. Data Description Markup Language (DDML)......40
6.1.1. Examples of type elements......41
6.2. Units in properties and events......41
6.2.1. Units for real values......41
6.2.2. Units for integer values......43
6.2.3. Unit compatibility......43
7. SIMULATION DESCRIPTION MARKUP LANGUAGE (SDML)......44
7.1. Specification of SDML......44
7.2. Simulation structure in SDML......44
7.3. Component and system initialisation in SDML......45
8. OTHER ELEMENTS OF PROTOCOL MESSAGES......46
8.1. Names......46

8.2. Registration identifiers....46
8.3. Message identifiers....46
8.4. Property and event matching....46

9. IMPLEMENTATION OF THE PROTOCOL....48

9.1. Layout of messages....48
9.1.1. Message header....48
9.1.2. Message data....48
9.1.3. Examples of arrays....48
9.2. Component descriptor routine....50

10. COMPONENT IMPLEMENTATION TECHNIQUES....52

10.1 Common implementation interfaces for Microsoft Windows....52
10.1.1. Interface for simulation design and construction....52
10.1.2. Component wrapper DLLs....53
10.1.3. Distributing the Simulation over more than one machine....54
10.2 Note on system implementations....56

11. REFERENCES....57

APPENDIX: DIMENSIONS AND SI UNITS....58

1. Introduction

The purpose of this document is to specify a **modelling protocol**, a modular framework that enables the sub-components of simulation models to be interchanged between different modelling software.

Simulation modelling in agriculture and resource management has now been under way for over 30 years (Brouwer & de Wit 1968; Freer *et al.* 1970). Very early in that history, the desirability of generic simulation tools was recognized (e.g. Beek & Frissel. 1973). Over time the following attributes have been recognized as desirable in a generic agricultural simulation tool:

Hierarchical Ecological and hence agronomic systems are *medium-number* systems. They contain too many entities to be treated as *small-number* systems that can be solved by differential-equation techniques; and they have too few entities to be treated as *large-number* systems that are amenable to treatment as statistical assemblages.

Current ecological theory suggests that the best way to analyze this kind of complexity is to take advantage of *organization* in these systems that arises from differences in the rates of different processes. This organization leads naturally to representations of reality that are hierarchically structured (O'Neill *et al.* 1986).

Modular Similar considerations lead to the separation of closely-interacting parts of a model system into discrete entities in the model code. Modularization has practical benefits, especially in allowing scientists in research teams to specialise in modelling one part of the larger system (McCown *et al.* 1996).

Configurable Once a simulation model is decomposed into sub-models, it becomes natural to arrange the sub-models in different configurations to reflect a range of different real-world situations (McCown *et al.* 1996).

Interchangeable Modular construction also permits the substitution of one representation of a process by another, depending on the needs of the modeller. This can be useful in comparing different representations of a process, or in configuring a simulation for efficient execution.

Interchangeability applies not only to sub-models, but also to modelling software. Ideally it should be possible to use the software implementing a model in conjunction with a range of different user interfaces for different purposes (e.g. Donnelly *et al.* 1997).

Mixed discrete & continuous Many processes in agricultural systems are fundamentally continuous in nature. Others, particularly management interventions, involve sharp changes in the state of the system. Event-based representations of management have a long pedigree (e.g. Christian *et al.* 1978).

The protocol described in this document is intended to support the construction of simulation tools that meet these criteria.

This document describes version 1.0 of the modelling protocol.

1.1. Formal definition of a simulation[1]

Before defining a modeling protocol, it is important to define the kind of models - "simulation" models - that it is intended to support.

- A **simulation** is a computation of a **dynamic model** between given start and end times, i.e. it is an integration over time.
- A dynamic model is defined by a set of equations. The equations of a dynamic model may fall into natural groupings known as **submodels**. Some of these submodels may have equations and quantities of identical form, i.e. they belong to the same **submodel class**. The dynamic model as a whole can therefore be viewed as a collection of instances of various submodel classes.
- A submodel is composed of a set of **quantities**, a set of **rate equations**, and a set of **events**.
- All quantities can be expressed as real numbers, integers, or Boolean values. Real-valued quantities have **dimension** and **units**; the units must conform to the dimension. I identify different kinds of quantities:
  (a) **Constants** are quantities that are (i) invariant in time and (ii) have the same value in all instances of a submodel within a model and all simulations of a model.
  (b) **Parameters** are quantities that are invariant in time, but may take different values between different instances of a submodel within a model or between simulations of a model.
  (c) **State variables** are quantities that may vary in time as the simulation is computed. The value of a state variable must be stored in order to compute the dynamics of the submodel. As a result, the initial value of each state variable must be specified in order for the simulation to be computed. There is a one-to-one correspondence between state variables and the rate equations of the submodel. In principle, there should be no redundancy in the state variables.
  (d) **Summary variables** may also vary in time, but their value at any given time may be determined from the current values of the state and driving variables. They may be used to provide output from the simulation; to provide driving variables for other submodels; or as notational conveniences in the specification of the submodel's rate equations (in which case I refer to them as "intermediate" variables).
  (e) **Driving variables** are quantities which are stored externally to a given submodel but which must be known in order to compute the dynamics of the submodel. They may (and usually do) vary in time. Each driving variable must have a **source**, to which it is constrained to be equal at all times; a source may be a constant, parameter, state or summary variable from another submodel, or it may be a quantity external to the simulation. The set of submodel driving variables with external sources is the set of driving variables for the model as a whole.

  It should be noted that this terminology is not standardized; for example, "parameter" is used to mean (a), (b) and (c) above.
- Each real-valued state variable has a **rate equation** associated with it. The rate equation is an ordinary differential equation that gives the rate of change of that state variable over time. The right-hand side of each rate equation must be composed only of constants, parameters, state variables, summary variables and driving variables proper to the submodel.
- Each submodel has zero or more **events**[2] associated with it. An event, in this sense, is
  - a set of equations defining an instantaneous change in one or more state variables; and
  - a "trigger": a logical relation that, if satisfied at any time, causes the change(s) in state variables.

  Each event has zero or more quantities, known as **event parameters**, that may be used in specifying the right hand sides of the equations and the trigger along with constants, parameters, state variables, summary variables and driving variables proper to the submodel.
- A simulation is therefore completely defined by:
  - the model, i.e. the set of submodels it contains;
  - the start and end times for the computation;

---

1 No consistent terminology exists within the discipline of simulation modelling. The definitions made here are in relatively common use.

2 The term "event" is used in another sense within the modelling protocol. Model events will be represented by protocol events, but so will other parts of the computation. Unfortunately no good alternative term exists.

- the values of the state variables and event parameters of each submodel at the start time;
- the time course of the model's driving variables.

1.2. Roles involved in defining and using simulations

Simulation user	Uses a pre-configured dynamic model to execute simulations. The user specifies the data to be used during the simulation (e.g. the start and end times for the simulation, descriptions of soils, management rules, etc.) and interprets outputs.
Modeller	Configures sub-models to make a dynamic model. Verifies that the dynamic model is complete and scientifically meaningful. The modeller and simulation user may be the same person at the same time.
Component builder	Implements the quantities, rate equations and events of a submodel as a **component** (see section 2), using the methods provided by the protocol to communicate with the rest of a simulation.
Protocol implementer	Implements a version of the protocol for a particular operating system.

2. Definition of entities within the protocol

2.1 Components

A **component** is the entity within the protocol that encapsulates a submodel. The interface of a component is made up of:

- a name. When a component belongs to a system, the simulation can refer to it by a fully qualified name that includes that of it's parent system.
- a unique ID, used to denote the component in messages.
- properties (as defined below).
- event handlers (as defined below).

2.2 Properties

Properties encapsulate the quantities of each submodel. The interface of a property includes:

- a name.
- an ID, used to denote the property in messages. ID values for properties are unique within components so that the pair (component ID, property ID) uniquely identifies a property within the simulation.
- a type. The type of a property determines the set of values it may take and the units of those values, where applicable. The type must be either one of a set of **primitive types** (see Figure 2.1 and section 6.1) or else an array or record structure ultimately composed of these primitive data types.
- a value.

Two distinct kinds of properties are identified. **Driving properties** encapsulate driving variables, i.e. quantities which are stored externally to a given component but which must be known in order to compute the component's logic. All other properties (i.e. those that are stored by the component) are **owned properties.**

Owned properties must have at least one of the following attributes:

Writeable Other components may request that the value of a writeable property be changed.

Readable Other components may request the value of a readable property, i.e. other components have read access to these properties.

Driving properties may receive zero or more values from different components. A component determines whether the number of values returned to it by the rest of the simulation is valid.

Driving property registrations follow the registration scheme that is used for events as outlined in the next section.

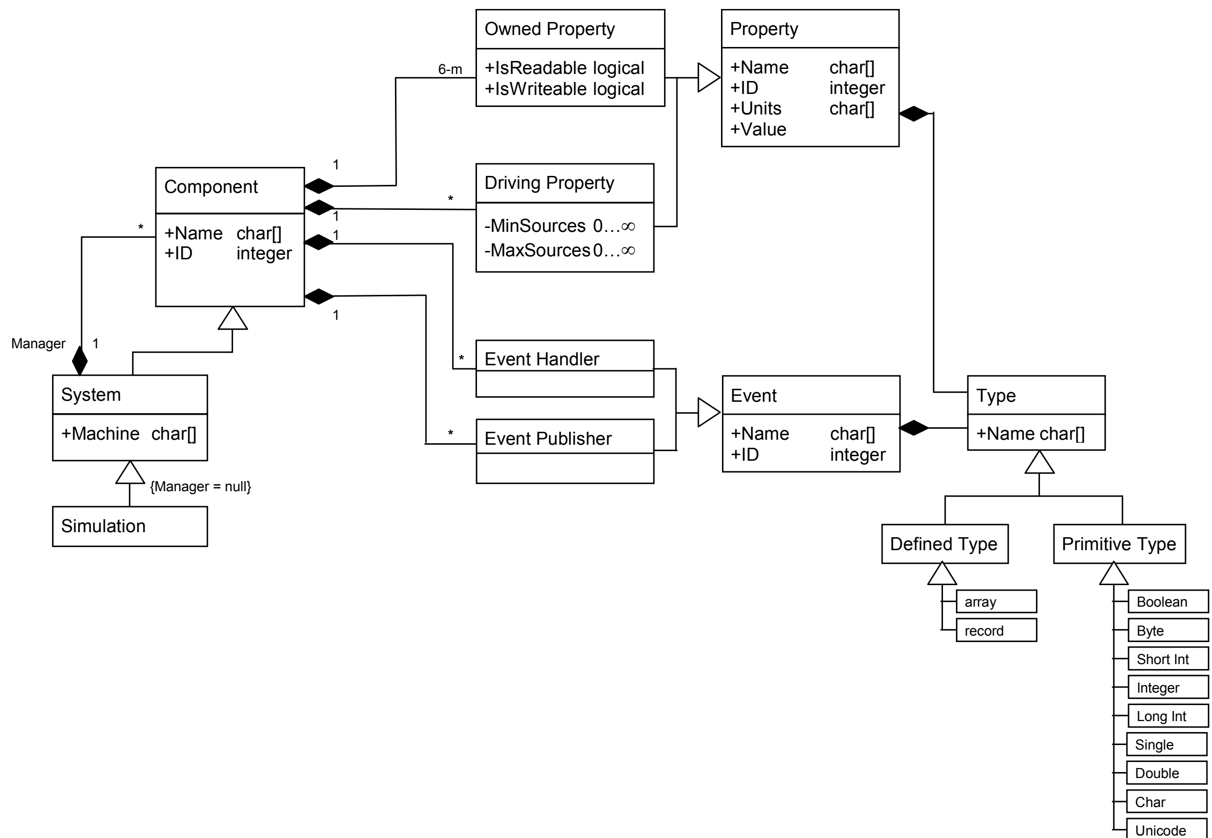


Figure 2.1. *Class diagram describing the relationship between entities in the modelling environment*

2.3 Events and event handlers

Events are used to signal the occurrence of activities (computations) and pass instructions between components. Events have:

- a *name*.
- a *type*. All events must have a record type; consequently each parameter of the event takes a name.
- *data* containing the values of the event's parameters.

Event handlers in the protocol encapsulate component logic (i.e. all computations that alter the state variables of the submodel encapsulated by the component). The interface of an event handler includes:

- a name.
- an ID, used to denote the event handler in messages. ID values for event handlers are unique within components so that the pair (component ID, handler ID) uniquely identifies an event handler within the simulation.
- a type. The type of an event handler is the same as the type of the data within the events that it handles.

Components may register event publishers in one of three ways:

Unqualified or partly-qualified name	Events published will go to any event handler that matches the name and type.
Fully-qualified name	Events published by this handler will go to the unique event handler that matches the name and type, assuming that it exists.
Unqualified name + integer component ID	The component ID is read as a destination for events. Events published will go to the unique event handler denoted by the component and handler name, assuming that it exists and has a matching type.

- It is an error to give both a component ID and a qualified name.
- It is always permitted to register an event publisher with no matching event handler. If the sender wants to ensure that the event is handled, it must request acknowledgement and count the **complete** messages as they come back.

2.4 Systems and simulations

A **system** is a component that groups related components within a simulation. In addition to the usual attributes of components, a system has:

- zero or more components within it who may be systems.

All components of a simulation are implemented on a single machine and in a single address space (except sub-systems, for which this is optional).

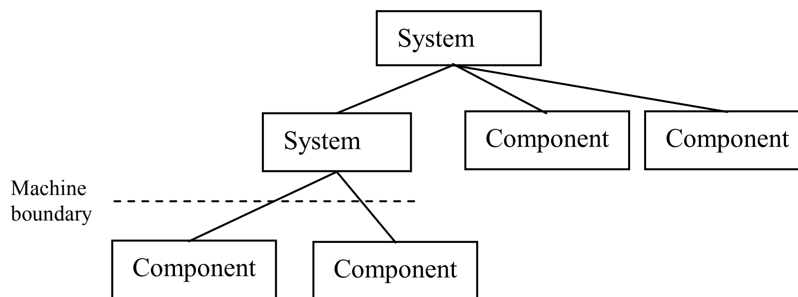


Figure 2.2 *An example of a typical structure containing systems and components*

A **simulation** is the execution of a model within the protocol. A simulation is equivalent to its top-level system; when its behaviour as a system is under consideration, it will be referred to as the **simulation system**.

Every component - except the simulation system - is a member of exactly one system. Systems are therefore arranged in a hierarchy or tree, with the simulation system at the root. The system that contains a component performs a number of tasks relating to that component; it is referred to as the system that **manages** the component.

2.5 Messages

Messages are the means by which information and requests are passed between components and systems as a simulation is computed. Messages are entities that can contain events; events are passed within messages. There is a defined set of 30 messages, each of which has a specific set of data defined that compose the message (see sect. 4). A component that receives a message may execute some of it's own internal logic (which may result in the component sending further messages); or it may be required to send particular messages as a mandatory response.

Messages that are sent from components are firstly received by the owning system. This system is then responsible to route the message to its owner or one of the other child components.

Sections 3 and 4 of this document describe the set of messages and the way that components use messages to carry out all the tasks necessary to execute a simulation successfully.

3. Tasks performed using the protocol

In order to support the execution of dynamic models, the protocol must carry out the following set of tasks:

1. Initialisation of a simulation
2. Termination of a simulation
3. Computation of a time step
4. Transmission of current values of a driving property
5. Changing the value of another component's owned property
6. Transmission of an event
7. Transmission of an error message
8. Recording the current model state (“checkpointing”)
9. Registration by a component of a property or event.
10. Removal by a component of a property or event registration
11. Addition of a component to a system
12. Removal of a component from a system
13. Deactivation of a component within a system
14. Activation of a component within a system
15. Obtaining information about components, properties or events

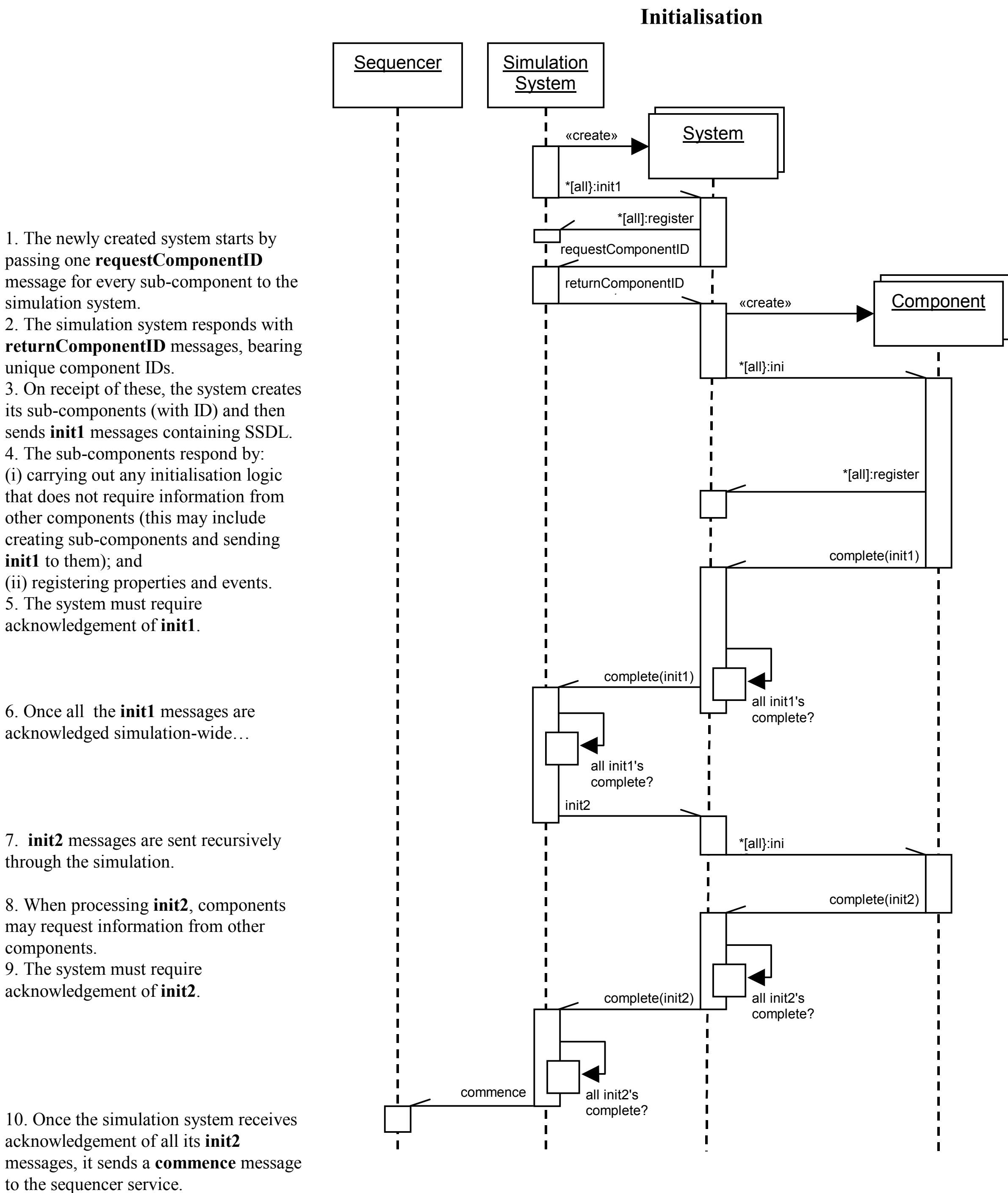
Each task is carried out by means of a sequence of messages between components and/or protocol managers. Sequence diagrams for each task are given in sections 3.1-3.10. The contents of the messages are set out in section 4 of this specification.

3.1. Initialisation of a simulation

Protocol implementations will support three different sources for initialisation information:
(a) the simulation user via an interface;
(b) the component itself, i.e. default values may be used; and
(c) other components in the simulation.

To support this, two distinct stages are employed in the initialisation process. A single-stage process is not feasible because of the need to allow initialisation information to be taken from other components

The constructor for a component includes the parameters; ID and Parent ID. This information is needed at construction time so that the **init1** message can be directed correctly.



Notes:

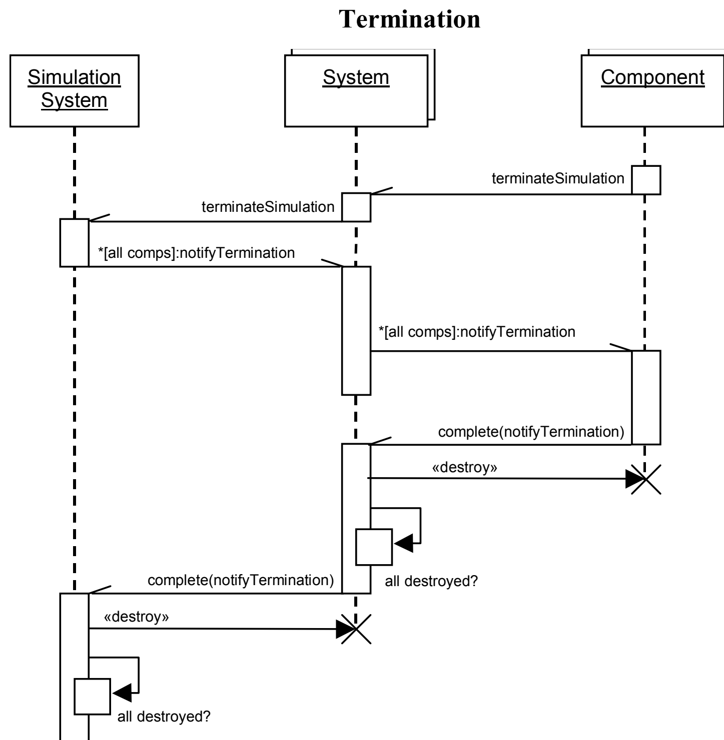
- This sequence diagram describes the initialisation of a system directly contained within the simulation system. The initialisation process operates recursively across the tree of systems that forms the whole simulation. The diagram also assumes that control of time-step execution has been delegated to a "sequencer" component.
- The simulation system (top-level system) must be created before the initialisation process can be initiated.
- All user-provided initialisation information is provided to a component in a single message as fragment of SDML. The fragment must conform with the <component> element in the SDML definition set out in section 7.1.
- A sequencer component is guaranteed to be ready to receive the **commence** message. In the absence of a sequencer, the simulation system must submit the **commence** message to itself.
- A component must know its ID before a message can be routed to it. Components are therefore passed their ID values as part of the creation process, not via a message.
- Component IDs must be unique throughout the simulation, so that the destination of every message acknowledgement is unambiguous. The simulation system is therefore given the responsibility of allocating IDs for all components.
- After components register their published events and driving variables, their owning system will, during init2, need to do queryInfo messages to find all the connections for the published events and driving properties.

Messages used:

requestComponentID	**returnComponentID**	**init1**	**init2**
REGISTER	**commence**	**complete**	

3.2. Termination of a simulation

1. Any component initiates termination by issuing **terminateSimulation** to the simulation system.
2. The t**erminateSimulation** message is passed back to the simulation system.
3. It sends a **notifyTermination** message to all its components.
4. Components respond to **notifyTermination** by performing any final processing.
5. Systems must also send **notifyTermination** to the components that they manage.
6. On acknowledgement of each **notifyTermination** message, the system destroys the acknowledging component.
7. Systems only acknowledge **notifyTermination** once all their sub-components have been deleted.
8. At the end of the process, the simulation system remains without any components.



Notes:

- Termination of a simulation is similar to a series of component deletions. However when a simulation is being terminated, the **pause** messages used in component deletion might be left without a destination and so not be acknowledged. Termination is therefore described as a distinct use case.

Messages used:

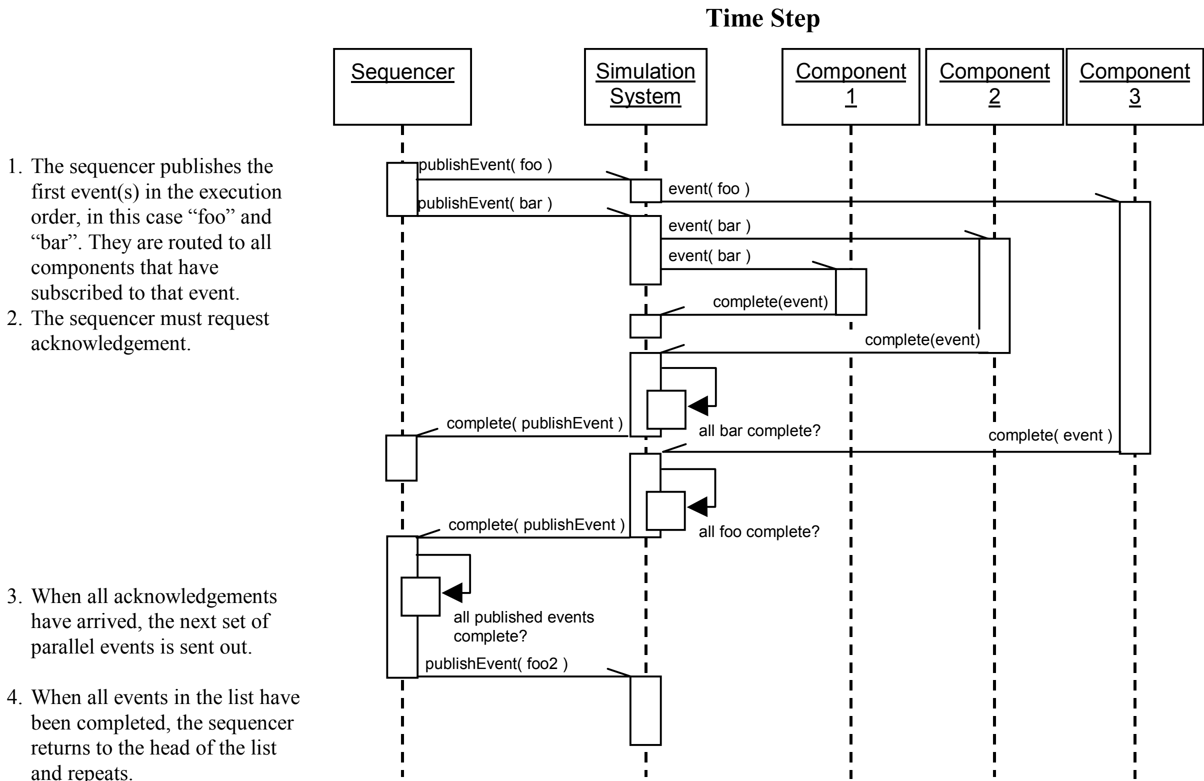
terminateSimulation **notifyTermination** **complete**

3.3. Computation of a time step

Every simulation must have a component that provides the *sequencing* service. This sequencer component registers a set of event publishers (the *sequenced events*) that instruct other components to carry out the computations that together constitute the integration of the simulation over a time step.

The sequenced events are ordered, in the sense that each given sequenced event is always sent before, parallel with, or after each other sequenced event as a timestep is computed.

The *Execute Phases* task is an endless loop that is started in response to a **commence** message. An iteration of the loop will typically correspond to a time step.

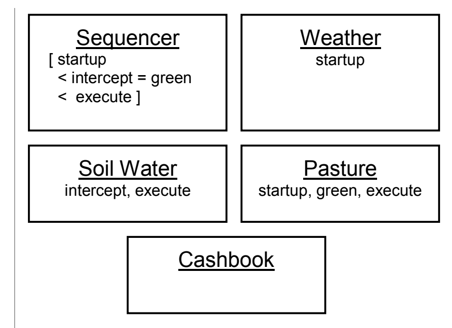


Messages used:

publishEvent **event** **complete**

As an example of how this scheme works, consider the simulation on the right. In this simulation, the sequencer will publish a total of four events each time step, in three ordered groups:

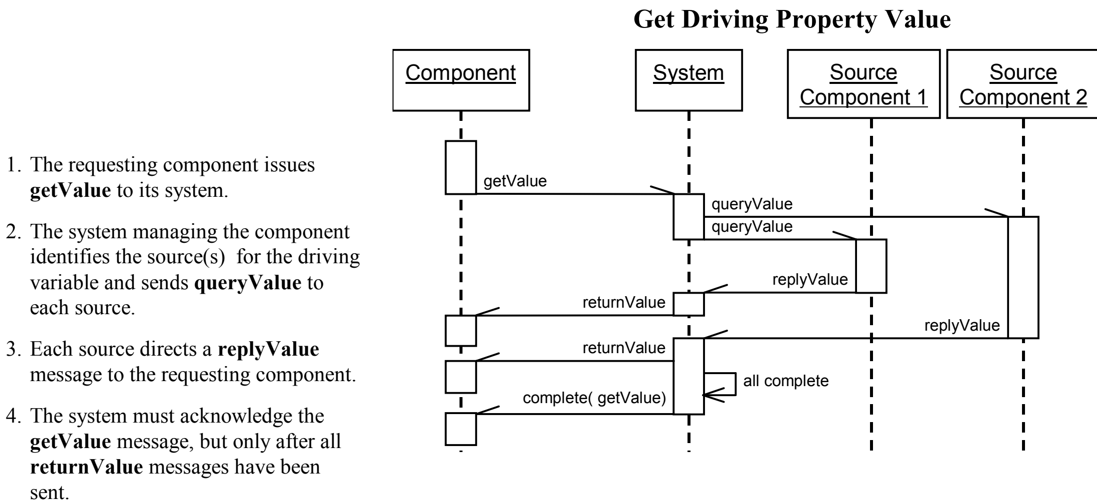
(a) “startup” to Weather and Pasture;
(b) “intercept” to Soil Water, at the same time as “green” to Pasture;
(c) “execute” to Soil Water and Pasture.



The sequencer service has no interaction with the Cashbook component.

3.4. Transmission of driving property values

The driving properties of a component are those for which a value or values are obtained from another component. The protocol supports cases where zero, one or more than one values for a driving property are returned.



Notes:

- The above sequence shows the minimal behaviour required. The **returnValue** message(s) may be sent to other components that use the property as a driver.
- When the number of sources found by the system is out of the valid range for the property, the component must generate a fatal error.
- The arrival of a **queryValue** message for a property that is not readable causes a fatal error.
- When the source and destination components reside in different systems, the **queryValue** and **replyValue** messages are passed along the path of systems between the two managing systems. Because systems are nested, this path is unique.

Messages used:

getValue **queryValue** **replyValue** **returnValue** **complete**

Example:

Consider two components in the same system. **Comp1** has a driving property **x**, and **Comp2** owns a property **x**. **Comp1**, **Comp2** and the two **x** properties each have a registration ID. The protocol manager that lives within the system has worked out that **Comp2.x** acts as the only source for **Comp1.x**. Internally, **Comp1.x** is denoted by the pair (98,4) and **Comp2.x** by (99,33).

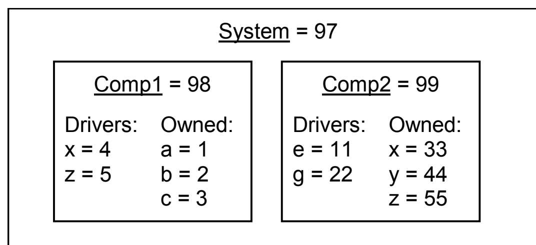


Figure 3.1. Example for getting driving property values

(a) When **Comp1** issues a request for the value of **x**, it sends

From=98 To=97
getValue(*id*=4)

(b) **System** receives this. It holds the driver-source relationship and so responds by sending

From=97 To=99
queryValue(*id*=33,
requestedby=98)

(c) **Comp2** receives this. It then addresses its answer to the system that made the **queryValue** request.

From=99 To=97
replyValue(*queryid*=msg ID of **queryValue**,
type="<type kind=string/>",
value="quick brown fox")

(d) **System** receives **replyValue**. It looks up the (previously stored) destination component and registration ID using the *queryid* field and sends **returnValue** message to **Comp1**.

From=97 To=98
returnValue(*compid*=99,
id=4,
type="<type kind=string/>",
value="quick brown fox")

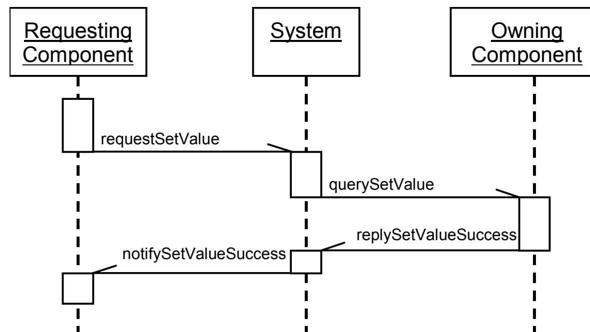
(e) **Comp1** receives this and can tell from *id* which driving property to assign the value to.

3.5. Changing another component's owned property

No component may directly change the value of another component's owned property. As a result the “writing” of owned property values must be implemented as a “request to change” that may be rejected by the receiving component.

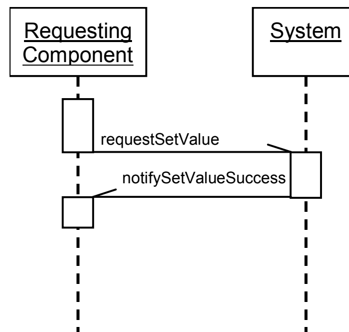
Alter Owned Property

(a) The property is present in the simulation



1. A **requestSetValue** message containing a previously registered ID is sent to the managing system.
2. The system identifies the component that owns the property to be altered and sends a **querySetValue** message to it.
3. The owning component responds with a **replySetValueSuccess** message.
4. The system converts this to a **notifySetValueSuccess** message informing the original sender whether the operation succeeded.

(b) The property is not present in the simulation



1. A **requestSetValue** message containing a previously registered ID is sent to the managing system.
2. The system responds with a **notifySetValueSuccess** message informing the sender that the property is not present.

Messages used:

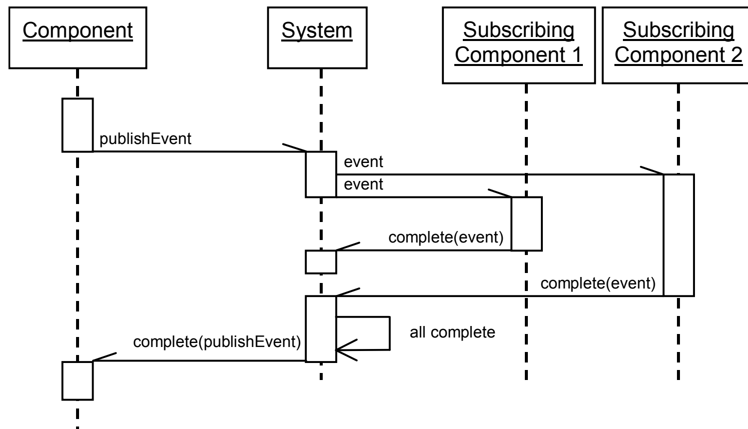
requestSetValue **querySetValue** **replySetValueSuccess** **notifySetValueSuccess**

Notes:

- If the name corresponding to the registration ID passed in the **requestSetValue** message is ambiguous, or if the property that it denotes is not writeable, the receiving system must generate a fatal error.

3.6. Transmission of an event

Transmit Event



1. A component sends a **publishEvent** message to its system. The system's PM sends **event** messages to all components that have subscribed to the event.
2. Acknowledgement is, in general, optional; if requested, it is routed in the usual way.
3. The **publishEvent** message is only acknowledged once the **event** messages have been acknowledged.

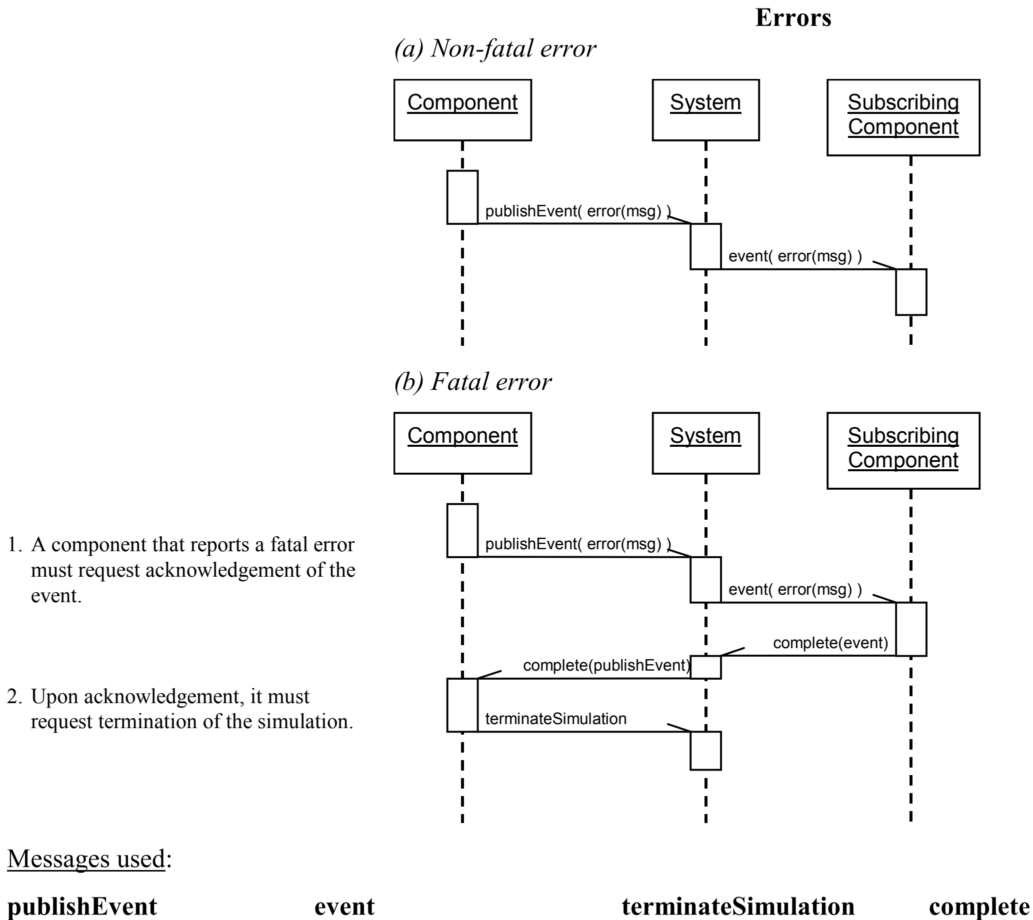
Messages used:

publishEvent **event** **complete**

3.7. Transmission of an error message

Transmission of error messages is handled as a standard event rather than as a distinct message. This permits a specific component or components to subscribe to the **error** event and act as, for example, an error-logger.

There are two use cases: one for a non-fatal (warning) error, and the other for a fatal error.



Messages used:

publishEvent **event** **terminateSimulation** **complete**

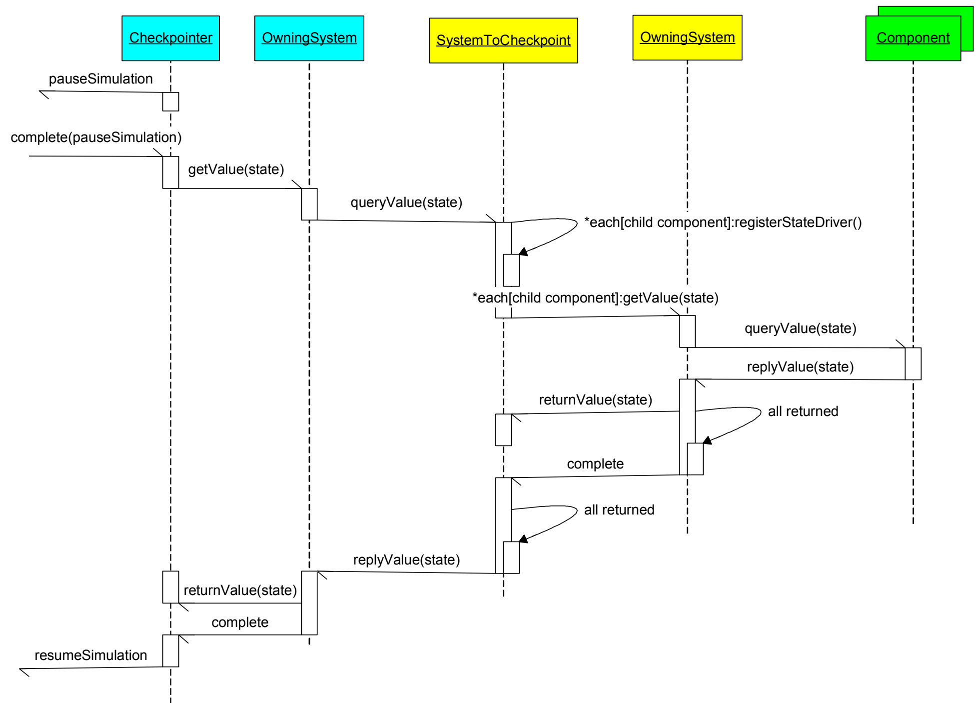
3.8. Recording and restoring the model state

3.8.1. Recording the model state

The process of recording the current state of a simulation, or part of a simulation, at some point during its execution is referred to as **checkpointing**. SDML fragments or scripts are used to record the current state of a component so they can be used to initialise the same component in another simulation.

Any component or system component may be checkpointed, including the simulation. Before a checkpoint is recorded, the simulation must be paused to ensure that the states of all checkpointed components are consistent. The checkpoint process involves using the sequence described in section 3.4. The **getValue** message is used to retrieve the standard **state** property of the component or the system that is being checkpointed. Since the definition for a **state** variable specifies that it includes any state information from child components, the system being checkpointed must have a special handler for **queryValue**(state) messages.

*Checkpointing using the **getValue** message.*



Once the simulation has been paused, the checkpointer uses a registered driving variable for the **state** property of the system being checkpointed. Then it issues a **getValue** for the **state** driver.

The **getValue** is passed to the parent of the checkpointer.

The checkpointed system uses driving variables for all it's child components to retrieve the **state** values of the children.

As each driver is returned the **state** value is constructed.

The **state** value is returned to the **checkpointer** and the simulation is resumed.

Messages used:

pauseSimulation	**getValue**	**replyValue**	**complete**
resumeSimulation	**queryValue**	**returnValue**	

Notes:

- The SDML fragment returned by each component as the value of the **state** property must be a valid `<component>` or `<system>` element in SDML as described in section 7 of this document. i.e. be capable of initializing the component.
- The driving variables used during the checkpoint process must be registered if they have not already been registered.

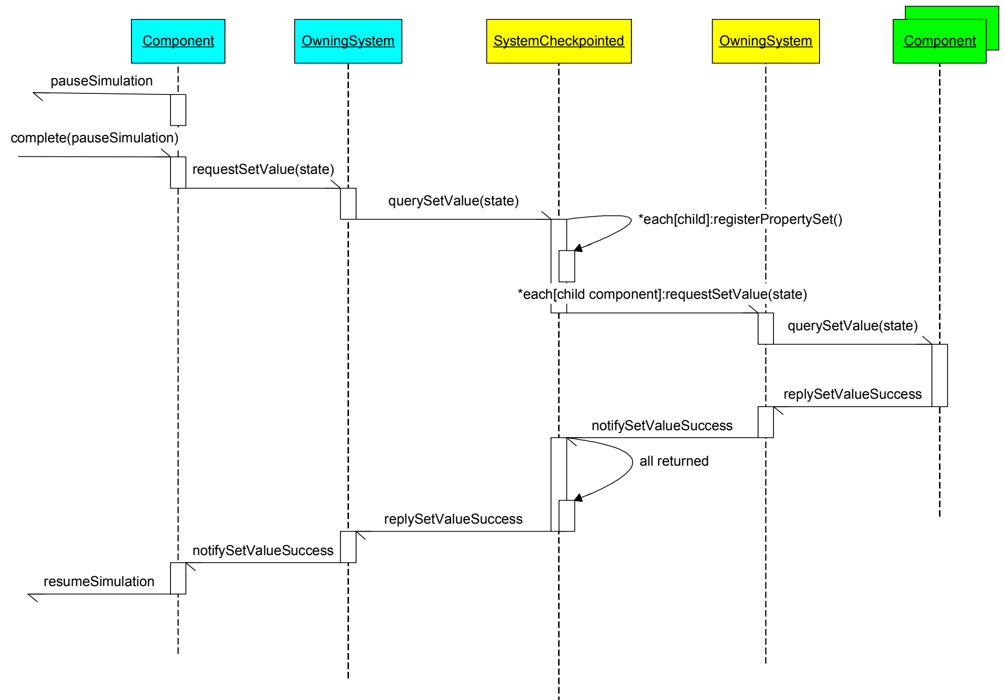
3.8.2. Restoring the model state

The process of restoring, or reinstating, the state of all or part of a simulation is carried out as the converse of the checkpointing process.

The process used to restore the **state** property is described in section 3.5 *Changing another component's owned property*. The SDML however can be used as the initialisation for the component in a new simulation script.

Since the definition for a **state** variable specifies that it includes any state information from child components, the system being restored needs to have a special handler for **querySetValue**(state) messages.

Restoring a checkpoint



The component pauses the simulation.

The component uses a registered property setter variable to send the **state** SDML to the system that was checkpointed.

The system component being restored may need to register property setters for it's children.

When the **notifySetValueSuccess** message returns, the component sends a **resume** message.

<u>Messages used</u>:

pauseSimulation	**requestSetValue**	**replySetValueSuccess**	**complete**
resumeSimulation	**querySetValue**	**notifySetValueSuccess**	

<u>Notes:</u>

1. If the structure of the sub system being restored is found to be different to the structure that was checkpointed, then a fatal error will be issued.

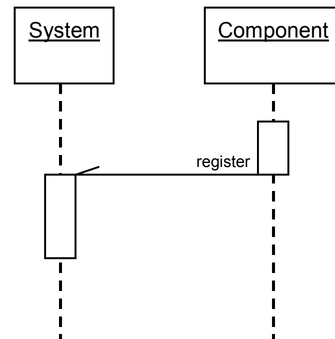
3.9. Changing the model structure

3.9.1. Registration of a property or event

Register Property or Event

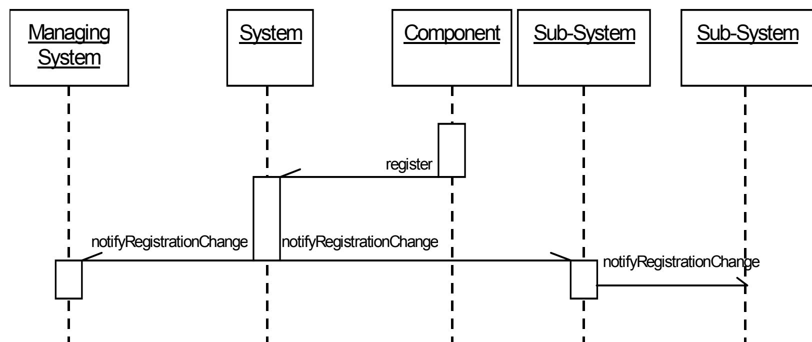
(a) During simulation construction

1. The requesting component sends a **register** message to its system.



(b) After simulation construction

1. After simulation construction, the system also sends a **notifyRegistrationChange** message to its parent system and to any sub-systems that it manages. This message propagates through the simulation.



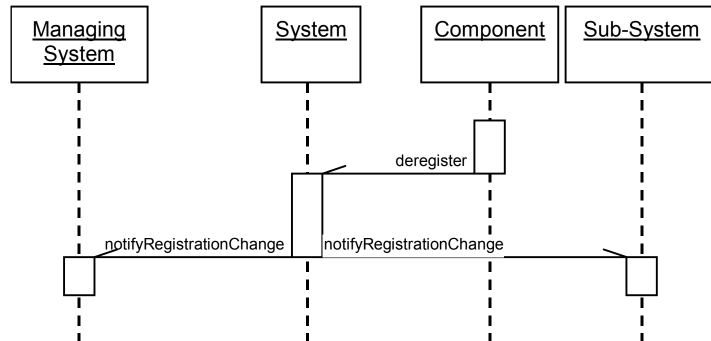
Messages used:

register **notifyRegistrationChange**

Notes:

- **notifyRegistrationChange** messages are propagated to every system in the simulation. This enables each system to keep track of the properties and event handlers to which it may need to route **queryValue**, **requestSetValue** and **event** messages.

3.9.2. Deregistration of a property or event



1. The requesting component sends a **deregister** message to its system.
2. The system records the deregistration.
3. It also sends a **notifyRegistrationChange** message to its parent system and to any sub-systems that it manages.

Messages used:

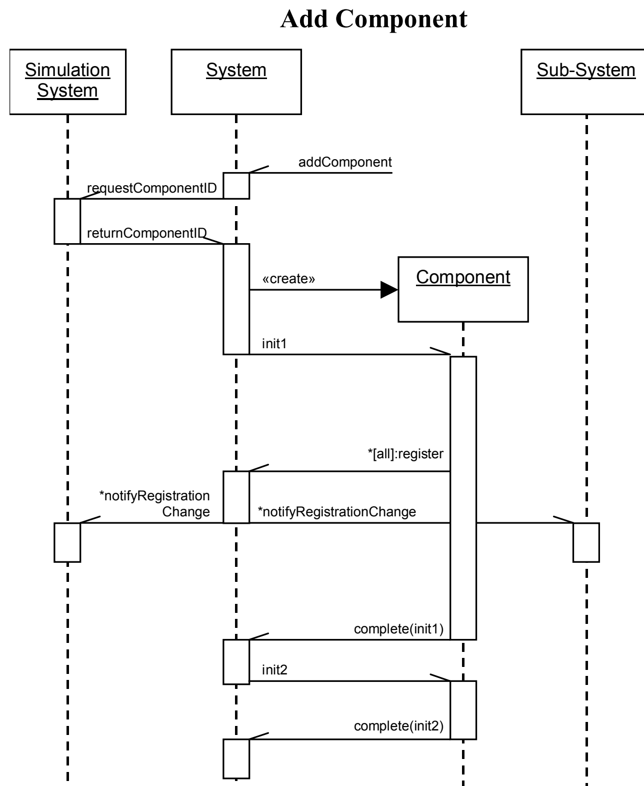
deregister **notifyRegistrationChange**

3.9.3. Adding a component to a system

Add Component

1. The system receiving **addComponent** passes a **requestComponentID** message back to the simulation system.
2. The simulation system replies with a **returnComponentID** message bearing a unique component ID.
3. On receipt of these, the system creates the new component (with ID) and then sends it an **init1** message containing SSDL.
4. The new component responds by:
(i) carrying out any initialisation logic that does not require information from other components; and
(ii) registering properties and events.
5. Component addition will normally take place after simulation construction, so **notifyRegistrationChange** messages are sent as part of registering each property and event.

6. The system requires acknowledgement of **init1**.
7. Once the **init1** message is acknowledged, the system sends an **init2** message.
8. The system requires acknowledgement of **init2**.



Messages used:

addComponent	**requestComponentID**	**returnComponentID**	**init1**
init2	**register**	**notifyRegistrationChange**	**complete**

Notes:

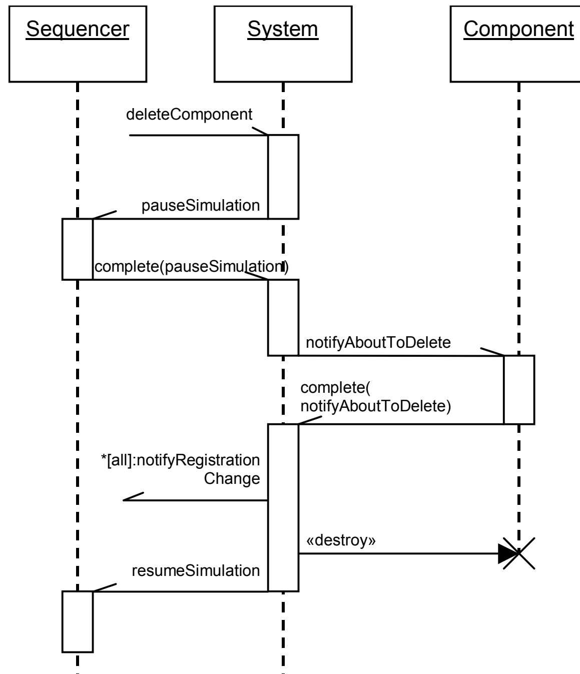
- Addition of a single component to a simulation bears similarities to the initialisation process. The main point of difference is that the **init2** message is sent immediately on receipt of the acknowledgement that **init1** has been processed; in the initialisation process, the **init2** cannot be issued until all the other components in the simulation have completed their first initialisation stage as well.
- Attempting to add a component to a component that is not a system causes a fatal error.

3.9.4. Removing a component from a system

Delete Component

1. The system owning the component to be deleted receives the message.

2. A **pauseSimulation** message is sent; acknowledgement must be requested.
3. Once the simulation is known to be paused, the system sends a **notifyAboutToDelete** message to the component. Acknowledgement is mandatory.
4. It responds by carrying out any cleanup logic & acknowledging.
5. The system then invalidates all registrations from the component. **notifyRegistrationChange** messages will be triggered.
6. Once the system has finished the deregistrsation, it deletes the component from memory.
7. The simulation is resumed.



Messages used:

deleteComponent **pauseSimulation** **resumeSimulation** **notifyAboutToDelete** **complete**

Notes:

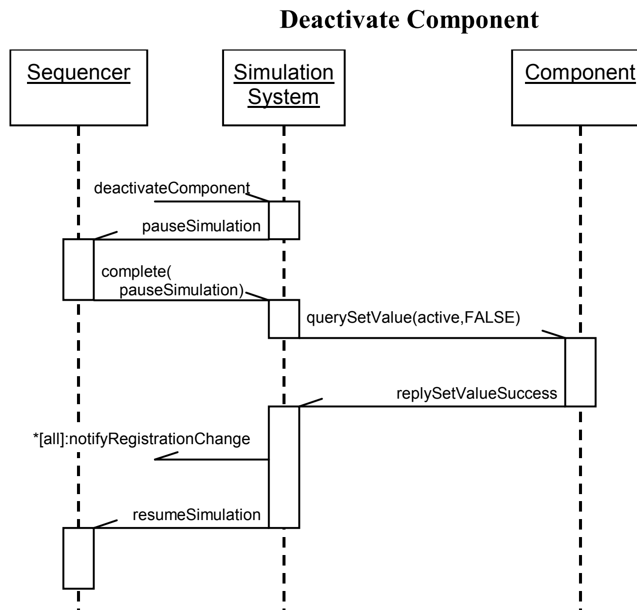
- The **notifyAboutToDelete** message is required to force the component to execute any final logic. Because this may involve communication with other components, it has to happen before deregistration.

3.9.5. Deactivating a component

All components possess a standard Boolean property named **active**. When the component is active, this property takes a value of TRUE. The *Deactivate Component* sequence diagram below describes the case where the **activity** property is TRUE. If it is already FALSE, no changes to registrations will be made.

Deactivate Component

1. The **deactivate** message arrives at the system managing the component (in this case, the simulation system).
2. A **pause** message is propagated to the sequencer; acknowledgement must be requested.
3. Once the simulation is paused, the system sets the **active** property of the target component to FALSE.
4. Once this is complete, the system invalidates all registrations from the component, isolating it from the simulation. **notifyRegistrationChange** messages will be triggered.
5. A **resumeSimulation** message is then propagated to the sequencer service to re-start the simulation. This time acknowledgement is not required.



Messages used:

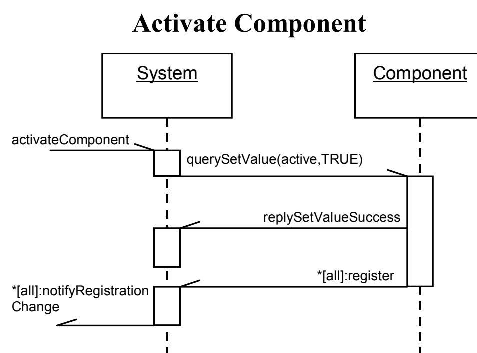
deactivateComponent **pauseSimulation** **resumeSimulation** **requestSetValue** **notifySetValueSuccess** **notifyRegistrationChange** **complete**

3.9.6. Activating a component

The *Activate a Component* sequence diagram describes the case where an **activate** message arrives at a component that has its **active** property set to FALSE. If it is already TRUE, no registration messages will be issued by the target component.

Activate Component

1. The **activate** message arrives at the system owning the component.
2. It sets the target component's **active** property to TRUE.
3. The component re-registers all the properties and events that were deregistered when it was deactivated. **notifyRegistrationChange** messages will be triggered.

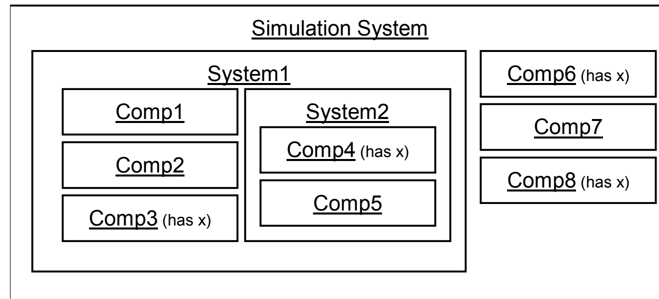


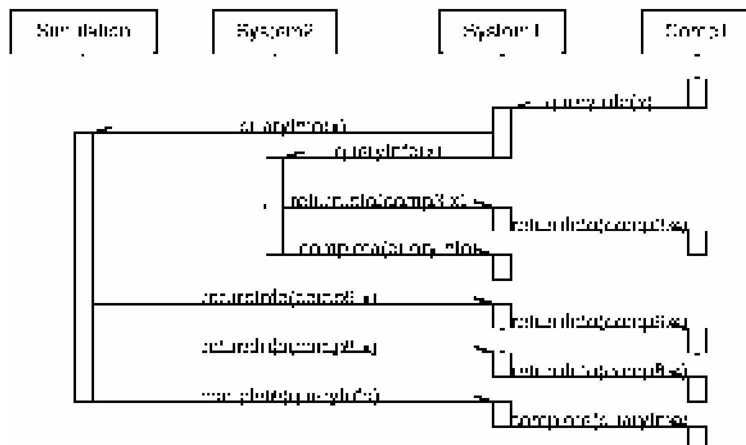
Messages used:

activateComponent **requestSetValue** **notifySetValueSuccess** **register** **notifyRegistrationChange**

3.10 Obtaining information about properties, components or events

This sequence diagram shows the flow of messages when information about a property or event (say "x") is enquired for in a simulation with the following structure:





1. The requesting component sends a **queryInfo** message to its system.
2. The system provides information about any matching entities using **returnInfo**…
3. and propagates the query to the system that manages it and any sub-systems.
4. These systems recurse the process.
5. Note how the simulation system responds with one message per matching entity.

<u>Messages used</u>:

queryInfo **returnInfo**

<u>Notes:</u>

1. When a child system receives a **queryInfo** it will not send it back to the parent that sent it.

4. Protocol Messages

4.1. Summary of protocol messages

The following is the proposed table of valid message types:

MsgType	Message Name	Sent by	Received by	Acknowledge Completion
1	**activateComponent**	Component	System managing component	Optional
2	**addComponent**	Component	System	Optional
3	*not used*			
4	**commence**	Simulation system	Simulation system (sequencer)	Optional
5	**complete**	Component	Component	Never
6	**deactivateComponent**	Component	System managing component	Optional
7	**deleteComponent**	Component	System managing component	Optional
8	**deregister**	Component	System managing component	Optional
9	**event**	Component	Component	Optional
10	**getValue**	Component	System managing component	Mandatory
11	**init1**	System	Component managed by system	Mandatory
12	**init2**	System	Component managed by system	Mandatory
13	**notifyAboutToDelete**	System	Component managed by system	Mandatory
14	**notifyRegistrationChange**	System	Systems	Optional
15	**notifySetValueSuccess**	System	Component managed by system	Optional
16	**notifyTermination**	Simulation system, system	System, component	Mandatory
17	**pauseSimulation**	Component	Simulation system (sequencer)	Optional[3]
18	**publishEvent**	Component	System managing component	Optional
19	**queryInfo**	Component	System	Mandatory
20	**querySetValue**	System	Component	Futile
21	**queryValue**	System	Component	Futile
22	**register**	Component	System managing component	Optional
23	*not used*			
24	**replySetValueSuccess**	Component	System	Optional
25	**replyValue**	Component	System	Optional
26	**requestComponentID**	System	Simulation system	Futile
27	**requestSetValue**	Component	Component	Futile
28	**resumeSimulation**	Component	Simulation system (sequencer)	Optional
29	**returnComponentID**	Simulation system	Component	Optional
30	**returnInfo**	System	Component	Optional
31	**returnValue**	System	Component managed by system	Optional
32	**terminateSimulation**	Component	Simulation system	Never

Notes:

- Anything sent by a component can be sent by a system.
- Acknowledgement of a message is "futile" when a component is required to respond to it by replying with a different message. For example, the **queryInfo** message must generate a **returnInfo** message in reply, so acknowledging this message is redundant.
- "(sequencer)" denotes that the simulation system is responsible for sending or handling a message but that this responsibility will usually be delegated to a sequencer service.
- Integer message fields are all 4 bytes in length.

[3] In some of the tasks discussed in section 3, completion of **pause** messages must be acknowledged.

4.2. Protocol messages in detail

This section contains a description of each protocol message, setting out:

- the characteristic structure for its message data component;
- which entities send the message and under what conditions it is sent;
- which entities receive the message and the response, if any, required of them.

Where a response is not otherwise specified, a system that receives a message must route it toward its destination. In this case the "From" field of the message must be preserved, so that when the message arrives at its destination the "From" field gives the original sender of the message.

Name:	**activateComponent**		
MsgType:	1		
Summary:	Reverses de-activation of a component (see the **deactivateComponent** message).		
Message data:	*component*	char[ ]	Unqualified name of the component to be activated. Refer to sect. 2.1
Sent by:	Components.		
Sent when:	At any time.		
Received by:	System managing a component.		
On receipt:	The receiving system must set the nominated component's **active** property to TRUE using the *Alter Owned Property* sequence diagram set out in section 3.5. If the nominated component was previously inactive, it must respond by re-registering its properties and event handlers.		

Name:	**addComponent**		
MsgType:	2		
Summary:	Adds a new component to the system managed by the receiving system.		
Message data:	*sdml*	char[ ]	A SDML fragment containing initialisation information for the new component. The fragment must conform to either the <component> or <system> element in SDML (see section 7.1).
Sent by:	Components.		
Sent when:	At any time.		
Received by:	Any system.		
On receipt:	The receiving system must respond by sending a **requestComponentID** message to the system that manages it (to itself if it is the simulation system). Sect. 3.9.3		

Name:	**commence**
MsgType:	4
Summary:	Requests the commencement of the first time step.
Message data:	(None)
Sent by:	Simulation system
Sent when:	During initialisation, once all **init2** messages sent by the simulation system have been acknowledged.
Received by:	Component that implements the sequencer service (may be the simulation system).
On receipt:	The component must begin to execute the *Execute Phases* sequence diagram set out in section 3.3.

Name:	**complete**
MsgType:	5
Summary:	Informs the receiving component that a message it sent earlier has been processed.
Message data:	*ack-id* integer Message ID of the message being acknowledged (section 9.1.1)
Sent by:	Components and systems
Sent when:	Whenever an entity receives a message with the acknowledgement flag set, it must respond by submitting a corresponding **complete** message once it has completed processing of the original message.
Received by:	Any component or system
On receipt:	The response depends on the type of the message being acknowledged and whether the message is received by the simulation system, another system or a component:

Message acknowledged	Received by	Response required
init1	Simulation system	If this is the last **init1** message acknowledged, then **init2** is sent to all components managed by the simulation system.
	Other system	If this is the last **init1** message acknowledged, then the **init1** message from the system that manages this system is acknowledged.
init2	Simulation system	If this is the last **init2** message acknowledged, then **commence** is sent (either to the simulation system or the sequencer service).
	Other system	If this is the last **init2** message acknowledged, then the **init2** message from the system that manages this system is acknowledged.
notifyTermination	All systems	(a) The component that sent the acknowledgement is deleted. (b) If this is the last **notifyTermination** message acknowledged, then the **notifyTermination** message from the system that manages this system is acknowledged.
getValue	Component	Signals to the component that all values have been returned in response to the query. The response is handled internally by the receiving component.
Other messages	Component	Handled internally by the receiving component

Name:	**deactivateComponent**
MsgType:	6
Summary:	Deactivates a component
Message data:	*component* char[ ] Unqualified name of the component to be deactivated
Sent by:	Components.
Sent when:	At any time.
Received by:	System managing a component.
On receipt:	The receiving system must set the nominated component's **active** property to FALSE using the *Alter Owned Property* sequence diagram set out in section 3.5. The system must then invalidate any registrations of that component's properties and event handlers and send out **notifyRegistrationChange** messages to inform the rest of the simulation that it has done so.

Name:	**deleteComponent**		
MsgType:	7		
Summary:	Informs the receiving system that it is to initiate removal of one of the components it manages.		
Message data:	*component*	char[ ]	Name of the component to be deleted
Sent by:	Component.		
Sent when:	At any time.		
Received by:	Any system managing a component.		
On receipt:	The receiving system must respond by sending a **pauseSimulation** message; once this is acknowledged, it must send **notifyAboutToDelete** to the component to be deleted.		

Name:	**deregister**		
MsgType:	8		
Summary:	Invalidates the registration of properties and/or events.		
Message data:	*kind*	integer	1 denotes a driving property 2 denotes a readable owned property 3 denotes a writeable owned property 4 denotes a readable, writeable and owned property 5 denotes an event publisher 6 denotes an event handler 7 denotes a component 8 denotes a system component 9 denotes a property set request
	id	integer	Registration ID of the property or event to be deregistered. This field should be 0 if the *kind* field denotes a component or system.
Sent by:	Component.		
Sent when:	At any time.		
Received by:	System managing the sending component.		
On receipt:	The receiving system invalidates the registration of the given property or event. If the *kind* field is 7 or 8, all property and event registrations for the component or system are invalidated. The receiving system must also send a **notifyRegistrationChange** message to (i) the system that manages it, if any, and (ii) any sub-system that it manages (including the component that sent the **deregister** message, if applicable). **notifyRegistrationChange** messages are not sent if the deregistration happens while the receiving system is engaged in processing an **init1** message.		

Name: **event**
MsgType: 9
Summary: Notifies a component that subscribe to an event that the event has occurred.
Message data:

id	integer	Registration ID of an event handler that subscribed to the event
publishedBy	integer	ID of the component that sent the **publishEvent** message that caused this **event** message to be created
type	char[ ]	DDML <type> element describing the type of the parameter data. The type of the parameters must be a record, so that each parameter has a name.
params	variant	Parameter data, laid out according to the *type* field.

Sent by: System or component
Sent when: (i) Sent when the system receives a **publishEvent** message from one of its components. One **publishEvent** message may result in zero or more **event** messages being sent.
The acknowledgement flag of an **event** message is only set if the triggering **publishEvent** message has its acknowledgement flag set.
(ii)A component may send an **event** message to another component's event handler at any time.

Received by: Any component.
On receipt: Handled according to the internal logic of the component.

Name: **getValue**
MsgType: 10
Summary: Passes request for a driving property value from a component to its system for routing
Message data:

id	integer	Registration ID of the driving property of the requesting component for which a value is requested.

Sent by: Components.
Sent when: At any time
Received by: System managing the sending component.
On receipt: The receiving system identifies all components that provide values for the driving property and sends a **queryValue** message to each.
Acknowledgement of the **getValue** message must be requested. The receiving system waits until it has routed all the corresponding **returnValue** messages to the component before sending the **complete** message. The acknowledgement thus signals to the component that all values have arrived in cases where a variable number of values may be returned.

Name:	**init1**		
MsgType:	11		
Summary:	Instructs a component to carry out the first part of its initialisation.		
Message data:	*sdml*	char[ ]	SDML <component> or <system> element containing initialisation information provided by the simulation builder. See section 7 for SDML.
	fqn	char[ ]	Fully-qualified name of the component being initialised
	inStartup	boolean	TRUE if the message is sent during simulation construction, i.e. the processing of **init1** messages as part of the initialisation of the simulation. (At this time, other components cannot be guaranteed to be in existence.) FALSE otherwise, including when it is sent as part of adding a component to the simulation.
Sent by:	system managing a component.		
Sent when:	In response to a **returnComponentID** message, immediately after the system has created the component.		
Received by:	Newly created component.		
On receipt:	The component carries out the first part of its initialisation, including processing of initialisation information in the *sdml* field and registration of all properties and events. The component cannot rely on any other component or property being present in the simulation while **init1** is being processed.		

Name:	**init2**
MsgType:	12
Summary:	Instructs a component to carry out the second part of its initialisation.
Message data:	(None)
Sent by:	System managing a component.
Sent when:	(a) During initialisation: if the system is the simulation system, when all **init1** messages sent by the system have been acknowledged; otherwise on receipt of an **init2** message. (b) During addition of a component: upon acknowledgement of the **init1** message sent to the component.
Received by:	Component.
On receipt:	The component carries out the second part of its initialisation, including obtaining initialisation information from other components.

Name:	**notifyAboutToDelete**
MsgType:	13
Summary:	Informs a component that it is about to be deleted
Message data:	(None)
Sent by:	System managing component
Sent when:	In response to **deleteComponent**, after the system has ensured that the simulation is paused.
Received by:	Component
On receipt:	On receipt of this message, the component must carry out any internal computation required before it is deleted (e.g. closing files). This message must always be acknowledged.

Name:	**notifyRegistrationChange**		
MsgType:	14		
Summary:	Broadcasts a change in the registration of properties and events to the rest of the simulation.		
Message data:	*registered*	boolean	TRUE if a property or event has been registered; FALSE if deregistered.
	kind	integer	1 denotes a driving property 2 denotes a readable owned property 3 denotes a writeable owned property 4 denotes a readable, writeable and owned property 5 denotes an event publisher 6 denotes an event handler 7 denotes a component 8 denotes a system component 9 denotes a property set request
	ownerid	integer	Registration ID of the component that owns the property or event (or the component itself if *kind*=7 or 8)
	id	integer	Registration ID of the property or event. Should be zero if *kind*=7 or 8.
	name	char[ ]	Unqualified name of the entity that is the subject of the notification.
	type	char[ ]	If the entity is a component: Text string giving the value of the component's **type** property (see section 5.1) If the entity is a property or event: DDML <type> element giving the type and units
Sent by:	System		
Sent when:	In response to a **register** or **deregister** message, except during simulation construction, i.e. the processing of **init1** messages as part of the initialisation of the simulation. (At this time, components cannot be guaranteed to be in existence.)		
Received by:	System managing the sender; also sub-systems managed by the sender		
On receipt:	The receiving system uses the contents of the message as necessary to restructure the routing of **queryValue** and **event** messages. The receiving system must also send a corresponding **notifyRegistrationChange** message to the system that manages it and to all sub-systems it manages (excluding the sender of the original message), thereby propagating the notification through the entire simulation.		

Name:	**notifySetValueSuccess**		
MsgType:	15		
Summary:	Notifies the receiving component whether or not a previously sent **requestSetValue** message was successful.		
Message data:	*id*	integer	Registration ID of the property for which an alteration was requested (the component ID is in the From field of the message).
	success	boolean	TRUE only if the owning component changed the property value as requested.
Sent by:	Component.		
Sent when:	In response to a **requestSetValue** message.		
Received by:	Component that sent the **requestSetValue** message.		
On receipt:	Handled internally by the component.		

Name:	**notifyTermination**
MsgType:	16
Summary:	Informs a component that the simulation is about to be terminated
Message data:	(None)
Sent by:	System managing component
Sent when:	The simulation system sends **notifyTermination** messages in response to **terminateSimulation**; other systems send it in response to **notifyTermination**.
Received by:	Component
On receipt:	On receipt of this message, a component must: (a) if a system, send a **notifyTermination** message to all components within the system, requiring acknowledgement; (b) carry out any internal computation required before the simulation is halted (e.g. closing files). This message must always be acknowledged. If the component is a system, it must not acknowledge the message until all its sub-components have acknowledged their **notifyTermination** messages.

Name:	**pauseSimulation**
MsgType:	17
Summary:	Signals that submission of sequenced events should halt until a **resumeSimulation** message is received.
Message data:	(None)
Sent by:	Any component.
Sent when:	At any time. Must be sent in response to a **deleteComponent** message.
Received by:	(a) Simulation system; (b) Sequencing component.
On receipt:	(a) The simulation system routes the **pauseSimulation** message to the component that provides the sequencing service (if not itself). (b) The component that provides the sequencing service responds by halting the sending of sequenced events until such time as a **resumeSimulation** message is received.

Name:	**publishEvent**		
MsgType:	18		
Summary:	Passes an event notification from a component to its system for routing		
Message data:	*id*	integer	Registration ID of the event publisher
	type	char[ ]	DDML <type> element describing the type of the parameter data. The type of the parameters must be a record, so that each parameter has a name.
	params	variant	Parameter data, laid out according to the *type* field.
Sent by:	Components.		
Sent when:	At any time		
Received by:	System managing the sending component.		
On receipt:	The receiving system identifies all components that have subscribed to the event and sends an **event** message to each. If acknowledgement is required, the message is only acknowledged once all the **event** messages triggered by the message have been acknowledged.		

Name:	**queryInfo**		
MsgType:	19		
Summary:	Broadcasts a request for information about a component, property or event handler.		
Message data:	*name*	char[ ]	Name of the component, property or event handler about which information is requested. The name may be qualified (see section 8.1). A property or event name may be "*", which matches all names ("component.*" is permitted).
	kind	integer	1 denotes a driving property 2, 3 and 4 denote an owned property 5 denotes an event publisher 6 denotes an event handler 7 and 8 denote a component or system 9 denotes a property set request
Sent by:	Component.		
Sent when:	At any time.		
Received by:	Systems.		
On receipt:	The receiving system checks entities of the nominated kind that are registered with it and sends a **returnInfo** message to the originating component for each entity that matches *name*. If the *name* field may match an entity outside the system, the receiving system must send a corresponding **queryInfo** message to the system that manages it. If the *name* field may match an entity within a sub-system managed by the receiving system (excluding the sender of the message), it must send a corresponding **queryInfo** message to the sub-system. In this way the query is propagated through the simulation.		

Name:	**querySetValue**		
MsgType:	20		
Summary:	Issues a request to set the value of another component's writeable (owned) property.		
Message data:	*id*	integer	Registration ID of the owned property to be set within the component to which the message is addressed.
	type	char[ ]	DDML <type> element giving the type of the value data
	value	variant	Value data, laid out in accordance with the type of the property
Sent by:	System		
Sent when:	In response to a **requestSetValue** message.		
Received by:	Component.		
On receipt:	The receiving component must respond with a **replySetValueSuccess** message to inform the sending system of the success or failure of the operation.		

Name:	**queryValue**		
MsgType:	21		
Summary:	Requests the current value of another component's owned property.		
Message data:	*id*	integer	Registration ID of the owned property within the component to which the message is addressed.
	requestedby	integer	Registration ID of the component that is requesting the value (i.e. issued the original **getValue** message)
Sent by:	System		
Sent when:	Sent when the system receives a **getValue** message from one of its components. One **getValue** message may result in zero or more **queryValue** messages being sent from the owning system. It is the responsibility of the sending system to ensure that all recipients own the nominated property.		
Received by:	Component.		
On receipt:	The component must send a **replyValue** message containing the requested value to the system that sent the **queryValue** message (given by the message's *from* field).		

Name:	**register**		
MsgType:	22		
Summary:	Registers a property or event handler with the system		
Message data:	*kind*	integer	1 denotes a driving property 2 denotes a readable owned property 3 denotes a writeable owned property 4 denotes a readable, writeable and owned property 5 denotes an event publisher 6 denotes an event handler 9 denotes a property set request
	id	integer	Identifier to be used in messages relating to the property or event handler. All property IDs within the component must be unique, as must all event handler IDs and property set request IDs.
	destID	integer	Optional integer ID for a destination component. • If *kind*=1 or 9, identifies a component that owns a property corresponding to the driving property or property set request being registered. Value requests for the driving property must be routed to the nominated component only. • If *kind*=5, identifies a component that subscribes to the event being published. When the event is published, it must be routed to the nominated component only. • If *kind* is 2, 3, 4 or 6, dest*ID* must be zero. If *destID*=0, this field is ignored and routing depends upon the protocol implementation.
	name	char[ ]	Name of the property or event handler. . If *kind*=1, 5 or 9 and *destID*≠0, the name must be unqualified (if not, a fatal error results.) In other cases the name may be fully or partly qualified.
	type	char[ ]	DDML <type> element giving the type and units of the property or event handler
Sent by:	Component.		
Sent when:	At any time. Sent as part of processing the **init1** message.		
Received by:	System managing the component.		
On receipt:	The receiving system registers the property. It also sends **notifyRegistrationChange** messages as detailed in the description of that message (except during simulation construction).		

Name:	**replySetValueSuccess**		
MsgType:	24		
Summary:	Issued in response Notifies the receiving component whether or not a previously sent **requestSetValue** message was successful.		
Message data:	*requestID*	integer	Message ID of the original **querySetValue** message
	success	boolean	TRUE i.f.f. the owning component changed the property value as requested.
Sent by:	Component.		
Sent when:	In response to a **querySetValue** message.		
Received by:	System that sent the **querySetValue** message.		
On receipt:	The receiving system must send a **notifySetValueSuccess** message to the component that originally sent a **requestSetValue** message.		

Name: **replyValue**
MsgType: 25
Summary: Provides the value of a component's property to a managing system for sending to the requesting component.
Message data:

queryid	integer	Message ID of the **queryValue** message to which this message is a response
type	char[ ]	DDML <type> element describing the type of the value data
value	variant	Value data, laid out in accordance with the type of the property.

Sent by: Component that owns a property.
Sent when: In response to a **queryValue** message
Received by: System
On receipt: When the managing system receives **replyValue** from a component, it sends (not routes) a **returnValue** message to the component that originally requested the value. The *from* field of the second message must contain the ID of the managing system. It may also send **returnValue** messages to other components in its system that have registered the driving property.

Name: **requestComponentID**
MsgType: 26
Summary: Requests a component ID from the simulation system.
Message data:

replyto	integer	Registration ID of the system that will manage the new component
name	char[ ]	Qualified name of the component (see below)

Sent by: System
Sent when: As part of processing an **addComponent** or **init1** message, before a component managed by the system is created.
Received by: System managing the sending system. If the new component is to be managed by the simulation system, the simulation system sends the message to itself.
On receipt: If the receiving system is not the simulation system:

it sends a **requestComponentID** to the system that manages it, with the *name* field further qualified (e.g. if System1 receives System2.x, it sends System1.System2.x). This ensures that a fully qualified name arrives at the simulation system.

If the receiving system is the simulation system:

it generates a unique component ID and sends a **returnComponentID** message to the system given by the *replyto* field.

The *replyto* field has to be provided separately from the *From* field in the message header because in a deeply nested simulation, the *From* field in the message arriving at the simulation system will not contain the ID of the system that initiated the process.

Name: **requestSetValue**
MsgType: 27
Summary: Issues a request to set the value of another component's writeable (owned) property.
Message data:

id	integer	Registration ID denoting the property to be set within the sending component.
type	char[ ]	DDML <type> element giving the type of the value data
value	variant	Value data, laid out in accordance with the type of the property

Sent by: Component.
Sent when: At any time.
Received by: Managing system.
On receipt: The receiving system must identify the component and property to which the *id* field corresponds. Three cases are possible:

(a) Zero destinations: the system must send the requesting component a **notifySetValueSuccess** message with *success*=true.
(b) One destination: the system must send a **querySetValue** message to the component that owns the property to be altered.
(c) More than one destination: the system must issue a fatal error.

Name:	**resumeSimulation**
MsgType:	28
Summary:	Signals that processing of sequenced messages may recommence.
Message data:	(None)
Sent by:	Any component.
Sent when:	At any time. Must be sent in response to acknowledgement of a **notifyAboutToDelete** message.
Received by:	(a) Simulation system; (b) Sequencing component.
On receipt:	(a) The simulation system routes the **resumeSimulation** message to the component that provides the sequencing service (if not itself). (b) The component that provides the sequencing service responds by re-commencing the sending of sequenced events (assuming that they have been paused).

Name:	**returnComponentID**		
MsgType:	29		
Summary:	Provides a registration ID for a component that is about to be created.		
Message data:	*fqn*	char[ ]	Fully qualified name of the component
	id	integer	Registration ID of the component. The value of *id* must be non-zero.
Sent by:	Simulation system		
Sent when:	In response to a **requestComponentID** message.		
Received by:	System that will manage the component when it is created		
On receipt:	The receiving system must create the component and then send it an **init1** message.		

Name:	**returnInfo**		
MsgType:	30		
Summary:	Provides a component with information about an entity within the simulation.		
Message data:	*queryid*	integer	Message ID of the **queryInfo** message to which this message is a response
	compid	integer	Component ID that owns the entity
	id	integer	Registration ID of the entity about which information is being returned
	name	char[ ]	Fully-qualified name of the entity
	type	char[ ]	If the entity is a component: Text string giving the value of the component's standard **type** property (see section 5.1). If the entity is a property or event: DDML <type> element giving the type and units of the entity
	kind	integer	1 denotes a driving property 2 denotes a readable owned property 3 denotes a writeable owned property 4 denotes a readable, writeable and owned property 5 denotes an event publisher 6 denotes an event handler 7 denotes a component 8 denotes a system component 9 denotes a property set request
Sent by:	System		
Sent when:	In response to a **queryInfo** message		
Received by:	Component (not necessarily managed by the sending system)		
On receipt:	Handled internally by the receiving component		

Name:	**returnValue**		
MsgType:	31		
Summary:	Provides the value of a component's property to another component.		
Message data:	*compid*	integer	ID of the component that owns the property value that is being returned in this message.
	id	integer	Registration ID of the property for which a value is being returned (the value sent as *id* in the **getValue** message that is being responded to).
	type	char[ ]	DDML <type> element describing the type of the value data
	Value	variant	Value data, laid out in accordance with the type of the property.
Sent by:	System that manages the component that originally issued a **getValue** message requesting the value of the property.		
Sent when:	In response to a **replyValue** message with a *queryid* field matching a previously dispatched **queryValue** message.		
Received by:	Component.		
On receipt:	The component that originally requested the value handles **returnValue** messages according to its internal logic.		

Name:	**terminateSimulation**
MsgType:	32
Summary:	Initiates termination of the simulation.
Message data:	(None)
Sent by:	Any component.
Sent when:	At any time.
Received by:	Simulation system
On receipt:	The simulation system is required to issue **notifyTerminate** messages to all components that it manages, and to delete each component after its **notifyTerminate** message has been acknowledged.

5. Standard Properties and Events

There is a set of properties that every component must possess (R49-54). Also, the sequence diagrams in section 3 assume the existence of a number of certain properties and events. Hence there is a need in the protocol for a set of "standard" properties and events.

5.1. Standard component properties

The set of standard properties is:

Name	Type	Readable	Writeable	Meaning
name	char[ ]	Yes	No	Fully-qualified name of the component
type	char[ ]	Yes	No	Name of the component class (component type)
version	char[ ]	Yes	No	Version of the component class
author	char[ ]	Yes	No	Author of the component class
active	integer	Yes	No	Zero denotes an active component; a positive value denotes the number of **activate** messages required to make the component active.
state	char[ ]	Yes	Yes	SDML <component> or <system> element describing the current state of the component or sub system.

- All components must own each of these properties.
- The value of the **name** property is set during creation of a component (it is provided by the **returnComponentID** message returned to its managing system).
- The values of the **type**, **version**, and **author** properties are determined by the <module> element of the SDML provided to the component at initialisation. The initial value of the **active** property is set by the "active" attribute of the <component> element.

5.2. Standard event handlers

The set of standard event handlers is:

Name	Type	Meaning
error	fatal : boolean message : char[ ]	Notifies the rest of the simulation of the occurrence of an error condition.

- The standard event is optional; however if a component publishes or subscribes to a standard event, it must have the type and meaning set out above.

5.3. Time property

As noted in section 1, time has a special status in simulations of dynamic models. As with all numerical integrations in time, computation of simulations is carried out using time intervals of finite length ("time steps"). Although time is inherently continuous, it is computationally more convenient if time is quantised.

These considerations are taken into account in the protocol via the standard **time** property. The following conditions apply to **time**:

- no more than one component in a simulation may own the **time** property (the sequencer); and
- it must take the record type set out below.

Field	Type	Meaning
startDay	integer	Day number of the start of the time step
startSec	integer	Seconds past midnight of the start of the time step (0-86399)
startSecPart	double	Fraction of a second of the start of the time step (0.0-1.0)
endDay	integer	Day number of the end of the time step
endSec	integer	Seconds past midnight of the end of the time step (0-86399)
endSecPart	double	Fraction of a second of the end of the time step (0.0-1.0)

Time steps are therefore represented in continuous terms, but time steps down to one second can be dealt with in a discrete fashion. A day number denotes the day, from midnight to midnight, that contains the Julian Day Number (as used by the astronomical community) with the same value. 9 October 1995 is day number 2450000.

The component that owns the **time** property must update its value once per time step. The values in the *startDay*, *startSec* and *startSecPart* fields after an update must equal the values in the *endDay*, *endSec* and *endSecPart* fields before that update.

6. Definition of data types

As discussed in section 3, the types of all property data and event parameters are passed with the values in text form using a data description language.

The protocol can denote data of the following types:

- primitive data types such as character, integer, floating point, Boolean;
- record structures containing sequences of smaller types;
- multi-dimensional arrays.

Records and arrays may be nested within each other. Multi-dimensional arrays are denoted as arrays of arrays.

XML (Extensible Markup Language) is a standard language definition technology which can be used to describe complex structures. The data description language has been implemented within XML and is therefore known as Data Description Markup Language, or DDML.

DDML is used in the **returnValue**, **publishEvent**, **event** and **returnInfo** messages.

6.1. Data Description Markup Language (DDML)

XML grammars are defined using document type descriptions. Here is the DTD for DDML:

```
<!ELEMENT type (field+|element)? >
  <!ATTLIST type name CDATA "" >
  <!ATTLIST type kind (integer1|integer2|integer4|integer8|single|double
                      |boolean|char|wchar|string|wstring|defined) "defined">
  <!ATTLIST type array (T|F) "F">
  <!ATTLIST type unit  CDATA "-">
<!ELEMENT field  (field+|element)? >
  <!ATTLIST field name CDATA "" >
  <!ATTLIST field kind (integer1|integer2|integer4|integer8|single|double
                       |boolean|char|wchar|string|wstring|defined) "defined">
  <!ATTLIST field unit  CDATA "-">
  <!ATTLIST field array (T|F) "F">
<!ELEMENT element  (field+|element)? >
  <!ATTLIST element kind (integer1|integer2|integer4|integer8|single|double
                         |boolean|char|wchar|string|wstring|defined) "defined">
  <!ATTLIST element unit  CDATA "-">
  <!ATTLIST element array (T|F) "F">
```

Note that the "ddml" fields passed in the data of various messages are not complete XML documents, but <type> elements from DDML, for example:

```
<type kind=”double” array="T" unit="mm" />
```

which denotes an array of double-precision numbers with units of mm.

The values of the "kind" attribute of the <type> element denote the following:

Type	Description
integer1	Signed 8 bit integer
integer2	Signed 16 bit integer
integer4	Signed 32 bit integer
integer8	Signed 64 bit integer
single	IEEE single precision floating-point number (32 bits)
double	IEEE double precision floating-point number (64 bits)
boolean	Boolean value
char	Character (8 bits)
wchar	Unicode character (16 bits)
string	String of characters
wstring	String of Unicode characters
defined	A record or array definition laid out in this type definition

A type or sub-type is an array if the <array> attribute equals "T".

In addition to conforming to the DTD, a document must follow these rules to be valid DDML:

1. If the **kind** attribute of a <type>, <field> or <element> element equals "defined" and its **array** attribute equals "T", then it must contain exactly one <element> element and no other elements. In this case the <element> element defines the type of the elements of an array.
2. If the **kind** attribute of a <type>, <field> or <element> element equals "defined" and its **array** attribute equals "F", then it must contain zero or more <field> elements and no other elements. In this case the <field> elements define the types of the members of a record.

DDML in general, and type names in particular, are case sensitive.

6.1.1. Examples of type elements

(a) 4-byte integer, no units

```
<type kind="integer4" />
```

(b) 2-dimensional array of temperatures

```
<type array="T">
   <element kind="double" array="T" unit="oC" />
</type>
```

(c) Named record type containing scalar and array fields

```
<type name="a_record">
   <field name="height"       kind="double"   unit="mm"   />
   <field name="apples"       kind="integer2" array="T"   />
   <field name="conductivity" kind="single"   unit="dS/m" />
</type>
```

(d) Character string (character array)

```
<type kind="string"/>
```

is equivalent to

```
<type kind="char" array="T"/>
```

(e) Named type: array of records

```
<type name="supplements" array="T">
  <element>
   <field name="name"   kind="string"/>
   <field name="dmd"    kind="double" unit="%"    />
   <field name="ME2DM"  kind="double" unit="MJ/kg"/>
   <field name="cp"     kind="double" unit="%"    />
  </element>
</type>
```

6.2. Units in properties and events

As described in sections 2 and 3, integer- and real-valued quantities in the protocol have units. Units are expressed as character strings. Text and Boolean quantities do not have units; “unit” DDML attributes in any messages describing variables of these types should be disregarded.

6.2.1. Units for real values

Only certain unit strings are valid for real-valued quantities. The set of valid unit strings is generated by the grammar set out below. The principles on which this grammar is based are as follows:

- In general, SI units are used. Non-SI metric units are also permitted.
- Units are constructed from a restricted number of "base" SI units by applying scaling prefixes and powers and by combination into products and ratios.
- Integer, decimal and rational representations of powers are permitted. Where an exact integer representation exists, it is used. Otherwise, where an exact decimal representation exists it is used (e.g. 0.75, not 3/4). All powers are positive; negative powers are denoted by representing the unit in the denominator of a ratio.
- Where a unit is a ratio, all terms in the denominator must follow all terms in the numerator.
- Either "`-`" or "%" (where appropriate) are used to denote all dimensionless quantities.

- The null or empty string is a valid unit. It is reserved to denote situations where the unit is unknown or any unit is acceptable.
- The grammar may only be extended in future by expanding the set of "base" units.

```
<unit>                ::= [<term>{'.'<term>}]{'/'<term>} | '-' | '%'

<term>                ::= [<scale>]<scalable-unit>['^'<power>]
                        | <non-scalable-unit>['^'<power>]

<scalable-unit>       ::= 'g'|'m'|'s'|'K'|'A'|'mol'|'cd'
                        |'rad'|'sr'|'Hz'|'N'|'Pa'|'J'|'W'|'C'|'V'|'F'|'ohm'
                        |'S'|'Wb'|'T'|'H'|'oC'|'lm'|'lx'|'Bq'|'Gy'|'Sv'|'kat'
                        |'t'|'l'|'min'|'h'|'d'|'y'

<non-scalable-unit>::= 'rad'|'sr'|'deg'|'ha'

<scale>               ::= 'p'|'u'|'m'|'c'|'d'|'D'|'h'|'k'|'M'|'G'|'T'

<power>               ::= <integer> | <decimal> | <integer>'/'<integer>

<decimal>             ::= [<digit>{<digit>}]'.'<digit>{<digit>}

<integer>             ::= <digit>{<digit>}

<digit>               ::= '0'…'9'
```

Notes:

- Unit strings are case-sensitive; whitespace is permitted (and ignored).
- Base units and their dimensions are as follows (see the appendix for more detail on dimensions and SI units):

`g`	gram	M	`Wb`	weber	$L^{2}\ M\ T^{-2}\ i^{-1}$
`m`	metre	L	`T`	tesla	$M\ T^{-2}\ i^{-1}$
`s`	second	T	`H`	henry	$L^{2}\ M\ T^{-2}\ i^{-2}$
`K`	kelvin	θ	`oC`	degree Celsius	θ
`A`	ampere	i	`lm`	lumen	I
`mol`	mole	n	`lx`	lux	$L^{-2}\ I$
`cd`	candela	I	`Bq`	becquerel	T^{-1}
`rad`	radian	–	`Gy`	gray	$L^{2}\ T^{-2}$
`sr`	steradian	–	`Sv`	sievert	$L^{2}\ T^{-2}$
`Hz`	hertz	T^{-1}	`kat`	katal	$T^{-1}\ n$
`N`	newton	$L\ M\ T^{-2}$	`t`	tonne	M
`Pa`	pascal	$L^{-1}\ M\ T^{-2}$	`l`	litre	L^{3}
`J`	joule	$L^{2}\ M\ T^{-2}$	`min`	minute	T
`W`	watt	$L^{2}\ M\ T^{-3}$	`h`	hour	T
`C`	coulomb	T i	`d`	day	T
`V`	volt	$L^{2}\ M\ T^{-3}\ i^{-1}$	`y`	year	T
`F`	farad	$L^{-2}\ M^{-1}\ T^{4}\ i^{2}$	`deg`	degree	–
`ohm`	ohm	$L^{2}\ M\ T^{-3}\ i^{-2}$	`ha`	hectare	L^{2}
`S`	siemen	$L^{-2}\ M^{-1}\ T^{3}\ i^{2}$			

- The "u" scaling factor denotes "micro" (10^{-6}). Other scaling prefixes have their usual meanings.
- The tokens "/" and "." are each used with two different meanings, but the meaning can always be determined from the following token.
- "%" must be used alone; for example, "%/d" is not a valid unit.

Examples of valid units for real variables:

`hPa`	scaled unit
`MJ/m^2/d`	ratio with two terms in the denominator
`/s`	no numerator
`kg^0.75`	or `kg^.75`
`m^1/3`	but not `m^1/2`, which is grammatically correct but should be given as `m^0.5`
`g.m/s^2`	but not `m/s^2.g` as numerator terms must precede denominator terms

6.2.2. Units for integer values

Any string constitutes a valid unit for an integer variable.

6.2.3. Unit compatibility

The units of two real values are **identical** if the same terms appear in the numerator and denominator. The order of the terms is not considered when assessing identity (e.g. "s.m" and "m.s" are identical units).

The units of two real values are **compatible** if they have the same dimension.

If an integer value has a unit that is a production of the above grammar, then the same rules for identity and compatibility apply as for real values. Otherwise, units of integer values are identical only if their unit strings are identical (case-sensitive) and are compatible only if they are identical.

7. Simulation Description Markup Language (SDML)

As described in section 3, simulations and the components they contain are initialised using information held in a text format. The simulation description language defined here gives the following information:

- the structure and interactions of the simulation, including the initial list of systems and components;
- the name of the simulation and each system;
- optionally, registrations for variables and events to be used in the simulation;
- information required to initialise the values of each component's properties.

The simulation description language has been defined within XML and is therefore known as Simulation Description Markup Language, or SDML.

7.1. Specification of SDML

The document type description is:

```
<!ELEMENT simulation (sdmlversion, (system|component)+) >
   <!ATTLIST simulation name CDATA "simulation" >

<!ELEMENT sdmlversion (#PCDATA)>

<!ELEMENT system (location?, executable?, initdata?, (system|component)*)>
   <!ATTLIST system name   CDATA #REQUIRED >
   <!ATTLIST system active (T|F) "T">

<!ELEMENT component  (executable, initdata?)>
   <!ATTLIST component name   CDATA #REQUIRED >
   <!ATTLIST component active (T|F) "T">

<!ELEMENT location   (#PCDATA) >
<!ELEMENT executable EMPTY     >
   <!ATTLIST executable name      CDATA #REQUIRED >
   <!ATTLIST executable version   CDATA    >

<!ELEMENT initdata   (#CDATA) >
```

Any component that can be system must use the <system> tag even when no child components currently exist.The contents of <location> and <executable> elements in SDML documents are case-insensitive. <initdata> elements are case-sensitive. The contents of "name" and "version" attribute values are case-insensitive.

7.2. Simulation structure in SDML

SDML denotes the structure of a simulation by means of nested <system> and <component> elements within the <simulation> element. An example is given in Figure 7.1. Each simulation has a name.

```
<simulation name="Example simulation 1"/>
  <sdmlversion>1.0</sdmlversion>
  <component name="sequencer"/>   … </component>
  <component name="weather"/>     … </component>
  <system name="paddock1"/> …
    <component name="water"/>     … </component>
    <component name="phalaris"/>  … </component>
    <component name="clover"/>    … </component>
  </system>
  <system name="paddock2"/> …
    <component name="water"/>     … </component>
    <component name="fescue"/>    … </component>
    <component name="clover"/>    … </component>
  </system>
</simulation>
```

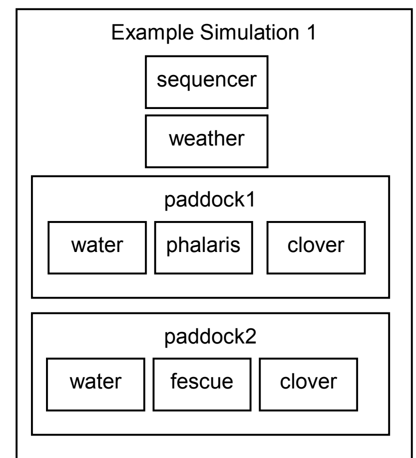


Figure 7.1. Representation of the structure of a simulation in SDML.

7.3. Component and system initialisation in SDML

Components are specified in SDML by means of <component> and <system> elements. These elements must contain:

- a "name" attribute that gives the name of the component; and
- an <executable> element that gives the name of an executable file containing the component logic.

They may also contain:

- an <initdata> element containing all initial values, in a format that is intelligible to the component but that is not specified within the protocol.

The <executable> element determines the value of a component's type as returned in a **returnInfo** message; for example all components using a module called "soilwater.dll" might have the type "Soil Water". Different modules may implement components of the same type.

A <system> element may also contain a <location> element that specifies a machine on which the system is to be executed. If this element is not given, the same machine as the containing system is used.

The "active" attribute specifies the initial value of the standard **active** property.

8. Other elements of protocol messages

8.1. Names

Simulations, components, properties and events have names that are used to denote them in certain messages and in SDML.

Names are composed of alphanumeric characters and the underscore ("_"). When names are compared, comparisons are case-insensitive.

Names may be "qualified" to reduce or eliminate ambiguity. Component names are qualified by preceding them by the name of the system of which they are a part, with a "." separator, e.g. "system.component". This process of qualification may be recursed, e.g. "system.subsystem.component".

Property and event names are qualified by preceding them by the name of the component of which they are a part, again with a "." separator. (The component name may itself be qualified, so that "system.component.property" is a valid qualified name.

A qualified name that begins with the name of a component managed by the simulation system is described as "fully qualified" as it has no possibility of ambiguity. The qualified name of a component is sent in the **requestComponentID** message, and fully qualified names for components are returned in the **returnComponentID** message when a component is being created. The standard **name** property returns a fully qualified name for the component.

"Checkpoints" also have names. These names may be any text string that can specify a storage location within the operating system in which the protocol is implemented.

8.2. Registration identifiers

Components, properties and event handlers also have integer ID values that are also used to denote them in messages. Component IDs are used as message addresses (see section 9.1). ID values are used in most messages for reasons of efficiency.

Each component is responsible for assigning ID values its properties and event handlers. Each property of a component must have a distinct registration ID, as must each event. It is permitted for a property to share its ID with an event, as the protocol messages always contain a context that makes the distinction clear.

The simulation system is responsible for assigning ID values to components. Each component in a simulation must have a distinct registration ID.

8.3. Message identifiers

Every message has an integer identifier that is assigned by the sender. Every message sent by a component must have a unique value for the identifier, so that the ordered pair (sender, message ID) identifies a message uniquely in the simulation.

8.4. Property and event matching

Where the sources for a driving variable or the subscriptions to an event are not specified by the simulation writer, the component's owning system must identify the properties or events in other components that match it. To match, two properties or events must have the same name (case-insensitive).

Regardless of whether sources or subscriptions are determined by the simulation writer or a system, the type of source properties or event publishers must also be **compatible** with the type of driving properties or event handlers. A type is compatible with another if a value of the first type contains all the information required to construct a value of the second type:

- Both types must be scalars, arrays, or records.

- If they are scalars, the following table applies:

First type \ Second type	int1	int2	int4	int8	sgl	dbl	boolean	char	wchar	str	wstring
int1	x	x	x	x							
int2	x	x	x	x							
int4	x	x	x	x							
int8	x	x	x	x							
sgl					x	x					
dbl					x	x					
boolean							x				
chr								x	x	x	x
wchar									x		x
str										x	x
wstring											x

  Also, the units of the two scalars (where applicable) must be compatible as defined in section 6.2.3.

- If they are arrays, then the type of elements of the first array must be compatible with the type of elements of the second array.
- If they are record structures, then for every field of the second type there must be a field in the first type that has the same name and a compatible type.
- A special case applies for record structures passed in the *params* field of **event** messages. In this case the type of the event data is compatible with the type of the handler only if:

  (a) for every field of the event handler type, either:
  - there is a field in the event data type that has the same name and a compatible type; or
  - the component implements a default value for the field

  and

  (b) every field in the event data type matches a field in the event handler type. Note that this condition prevents event handlers from taking a subset of the incoming data, as permitted in other contexts.

The final rule provides a mechanism for implementing events with default parameters. The component descriptor routine (section 9.2) provides the default values of event parameters.

Note that type compatibility is not a transitive relationship.

Protocol implementations may place further restrictions on the matching of properties and events, including further restrictions on type compatibility.

9. Implementation of the protocol

9.1. Layout of messages

Messages in the protocol are composed of a header and message data. The two components need not be contiguous in memory.

9.1.1. Message header

The header of all messages has the same structure, given in the following table:

Field	Denotes	Type
Version	Protocol version number. The high-order byte of this word denotes a major version number and the low-order byte a minor version number. This version of the protocol is 1.0.	2-byte word
MsgType	Unique ID denoting the type of the message. This must take a value from the table of message types in section 4.	2-byte word
From	Originating component, denoted by its registration ID.	4-byte integer
To	Destination component, denoted by its registration ID.	4-byte integer
MsgID	Message identifier (see section 8.3)	4-byte integer
Acknowledge	Flag denoting whether an acknowledgement message should be returned to the originating entity on completion of message processing (zero = no, non-zero = yes). This field has been given a length of 4 bytes to maintain word alignment.	4-byte word
NDataBytes	Number of bytes in the data component of the messages.	4-byte word
DataPointer	Pointer to the message data component. This pointer is NULL if NDataBytes is zero.	4 bytes

9.1.2. Message data

Each field of the message data is laid out sequentially in a contiguous block of memory. Message data fields are typed. The type of a data field must either be one of a finite set of primitive types, an array of a type, or a structure containing sub-fields (each with its own type).

Certain messages contain a field that denotes typed values. The data in these fields ("event" and "property" data within the "message" data) are laid out according to the same rules as for message data.

Data of primitive types occupy blocks of memory of the following sizes:

Type	Size (bytes)	Type	Size (bytes)
Boolean	1	Single	4
Byte	1	Double	8
Short Integer	2	ASCII Character	1
Integer	4	Unicode Character	2
Long Integer	8		

Text strings (ASCII or Unicode) are represented in message data as arrays of characters. They have been included in DDML for convenience.

Boolean values are False when the integer value of the field is zero.

Arrays are laid out as the number of array members (4 byte integer), followed by each array member concatenated together. All multi-dimensional arrays - including arrays of text strings - are represented as arrays of arrays.

Records are laid out with each member concatenated.

9.1.3. Examples of arrays

(a) One-dimensional array of integers: [5, 6, 7, 8]

Dim=4	5	6	7	8
4	4	4	4	4

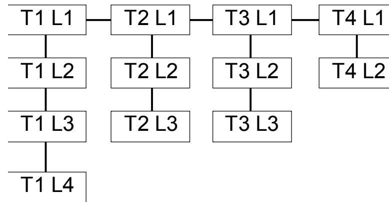
(b) Text string, represented as an array of character: "Quick"

Dim=5	"Q"	"u"	"i"	"c"	"k"
4	1	1	1	1	1

(c) Array of array of double-precision real values: $\begin{bmatrix} \begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix} & \begin{bmatrix} 40 \\ 50 \\ 60 \end{bmatrix} \end{bmatrix}$

Dim=2	Dim=3	10.0	20.0	30.0	Dim=3	40.0	50.0	60.0
4	4	8	8	8	4	8	8	8

(d) Array of leaf areas within an array of tillers:



Dim=4	Dim=4	T1 L1	T1 L2	T1 L3	T1 L4	Dim=3	T2 L1	T2 L2	T2 L3	Dim=3	T3 L1	T3 L2	T3 L3	Dim=2	T4 L1	T4 L2
4	4	8	8	8	8	4	8	8	8	4	8	8	8	4	8	8

(e) Array of text strings, represented as an array of array of character: ["Quick", "brown", "fox"]

Dim=3	Dim=5	"Q"	"u"	"i"	"c"	"k"	Dim=5	"b"	"r"	"o"	"w"	"n"	Dim=3	"f"	"o"	"x"
4	4	1	1	1	1	1	4	1	1	1	1	1	4	1	1	1

9.2. Component descriptor routine

The executable module that implements a component's logic (i.e. the executable referred to in the <module> element of the component's initial SDML) must have as part of its interface a function that returns a description of the component. The function is intended for use by the software that is used for writing simulations. The format for component descriptions is part of the protocol.

Component descriptions are XML documents conforming to the following DTD:

```
<!ELEMENT describecomp (executable, class, version, author,
                        system?, property*, event*)>

<!ELEMENT executable (#PCDATA)>
<!ELEMENT class      (#PCDATA)>
<!ELEMENT version    (#PCDATA)>
<!ELEMENT author     (#PCDATA)>
<!ELEMENT system     EMPTY    >

<!ELEMENT property (type)>
   <!ATTLIST property name   CDATA >
   <!ATTLIST property access (read|both|write|none) "read">
   <!ATTLIST property init   (T|F) "F" >

<!ELEMENT driver (type)>
   <!ATTLIST driver name    CDATA    >
   <!ATTLIST driver minsrc  CDATA "1">
   <!ATTLIST driver maxsrc  CDATA "1">

<!ELEMENT event (field*)>
   <!ATTLIST event name CDATA >
   <!ATTLIST event kind (published|subscribed) "published" >

<!ELEMENT type ( field+ | element+ | (defval,minval?,maxval?) )? )>
  <!ATTLIST type    kind (integer1|integer2|integer4|integer8|single|double|boolean
                         |char|wchar|string|wstring|defined) "defined">
  <!ATTLIST type    array (T|F) "F">
  <!ATTLIST type    unit  CDATA "-">
<!ELEMENT field  (name, ( field+ | element+ | (defval,minval?,maxval?) )? )>
  <!ATTLIST field   name CDATA "" >
  <!ATTLIST field   kind (integer1|integer2|integer4|integer8|single|double|boolean
                         |char|wchar|string|wstring|defined) "defined">
  <!ATTLIST field   array (T|F) "F">
  <!ATTLIST field   unit  CDATA "-">
<!ELEMENT element ( field+ | element+ | (defval,minval?,maxval?) )? >
  <!ATTLIST element kind (integer1|integer2|integer4|integer8|single|double|boolean
                         |char|wchar|string|wstring|defined) "defined">
  <!ATTLIST element array (T|F) "F">
  <!ATTLIST element unit  CDATA "-">

<!ELEMENT defval (#PCDATA)>
<!ELEMENT minval (#PCDATA)>
<!ELEMENT maxval (#PCDATA)>
```

- The <executable> element gives the location of the executable module providing the description.
- The <class>, <version> and <author> elements should return the values of the **type**, **version** and **author** standard properties.
- The <property>, <driver> and <event> elements give information about the component type's owned properties, driving properties, and event handlers respectively. Some component instances may have further properties and events defined at run-time; these are not included in the description.
- The "**access**" attribute of a <property> element denote whether the property is readable, writeable or both or neither. The "**init**" attribute denotes whether it may appear in the component's initializing SDML. “T” means that it is optional and “F” means that it will not appear.
- The "**minsrc**" and "**maxsrc**" attributes of a <driver> element give the range of replies to a request for that property which the component will accept. The "**minsrc**" attribute must denote a non-negative integer; the "**maxsrc**" attribute must either denote an integer greater than or equal to that denoted by the "**minsrc**"

attribute, or else be the null string (which means that the property has no maximum permitted number of sources). The integer values are parsed according to the rules for <defval> elements below.

- The type information for events is denoted as a list of fields so as to force each parameter of the event to take a name.
- The <type>, <field> and <element> elements closely follow the structures used in DDML.
- The optional <defval> elements provide default values. The interpretation depends upon the element in which they are found:

<property>	Default value for the property or property member. Particularly useful when the "init" attribute is set to "T".
<driver minsrc = "0">	Value that the component will use for the driving property in the absence of a source property.
<event kind = "subscribed">	Default values used when not all parameters of the event handler are transmitted in an event; see section 8.4.

In other cases, <defval> elements are ignored.

- The contents of <defval> elements for must follow the following rules:

`integer1`, `integer2`, `integer4`, `integer8`	Text must be an integer in decimal notation, i.e. a production from the following grammar:
	`<integer> ::= ['-'|'+']<digit>{<digit>}`
	`<digit>   ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'`

`single`, `double`	Text must be a real number in decimal or exponential notation, i.e. a production from the following grammar: `<real>    ::= <integer>['.'{<digit>}]['E'|'e'<integer>]`
`char`, `wchar`	Text must be a single character from the appropriate character set.
`string`, `wstring`	Any text from the appropriate character set is permitted.
`boolean`	Text must be a production from the following grammar: `<boolean> ::= 'TRUE'|'FALSE'|'true'|'false'`

- Numeric <property> elements may also give minimum and maximum legal values in <minval> and <maxval> elements respectively. Their contents must follow the rules for valid <defval> elements. These elements are ignored in other contexts.

10. Component Implementation Techniques

10.1 Common implementation interfaces for Microsoft Windows

In order to permit a component developed by one organization to be used by simulation software from other organizations, the following interfaces must be provided by protocol-compliant components implemented in Microsoft Windows. These interfaces should be regarded as a prescription for protocol implementations under other operating systems.

The `__stdcall` calling convention described here assumes that the exported function names are not mangled and do not have leading underscores. All functions below use the **stdcall** calling convention.

10.1.1. Interface for simulation design and construction

Protocol-compliant components must be implemented within Windows as dynamic link libraries. Each component executable must export the following functions.

(a) Component description interface

```
void getDescriptionLength(const char* szContext,
                                int* lLength )
```

Returns the length of the component description in bytes, excluding the final null character

```
void getDescription(const char* szContext,
                          char* szDescription )
```

Returns a null-terminated string containing the component description as set out in section 9.2.

The szContext parameter is designed to allow polymorphic components specify a description based on details within the character string. The contents of this string are implementation specific and if not used it should be an empty string.

(b) Initialisation script interface

As described in section 7, the initialisation information provided to a component in the SDML document (the "initialisation script" for each component) is in a format that is not known to the rest of the simulation. These routines form an interface for building and parsing the component-specific initialisation scripts from name, type and value data.

Initialisation scripts are understood to be composed only of initialised properties, each of which is composed of a name, a type and a value. Note the use of character strings to denote multiple instances of initialisation scripts for a particular component.

```
void createInitScript(const char* szScriptName )
```

Create a new initialisation script.

```
void deleteInitScript(const char* szScriptName )
```

Delete a previously created script.

```
void initScriptLength(const char* szScriptName,
                            int*  lLength      )
```

Length of a script in bytes, excluding the final null character

```
void textToInitScript(const char* szScriptName,
                            char* szScriptText )
```

Sets the contents of an initialisation script using a component specific format. The properties in this text will be appended to the list of any existing ones in this script.

```
void textFromInitScript(const char* szScriptName,
                        const char* szScriptText )
```

Returns the contents of an initialisation script

```
void valueToInitScript(const char* szScriptName,
                       const char* szPropertyName,
                       const char* szTypeDDML,
                             void* pValueData )
```

Sets an initial value within an initialisation script. `szTypeDDML` denotes the type using DDML; `pValueData` is a pointer to value data laid out as for message value data (section 9.1).

```
void valueFromInitScript(const char* szScriptName,
                         const char* szPropertyName,
                         const char* szTypeDDML,
                               void* pValueData )
```

Returns an initial value from an initialisation script. `szTypeDDML` denotes the type using DDML; `pValueData` is a pointer to value data laid out as for message value data (section 9.1).

(c) Name of the wrapper DLL

As described below, each component DLL has a "wrapper" DLL that implements an interface, between the simulation software and the component DLL, for passing messages.

```
void wrapperDLL( char* szWrapperDLL )
```

Returns the name of the "wrapper" DLL. If szWrapperDLL is null or zero length, the simulation assumes that the component DLL acts as the "wrapper"..

10.1.2. Component wrapper DLLs

As an aid to implementation, the common interface assumes that a "wrapper" DLL interposes between the simulation implementation and the component DLL proper. In all protocol implementations, internal component logic must be carried out by passing the message invoking the logic to the wrapper DLL; the wrapper DLL then passes the message contents to the logic DLL by implementation-specific means. A component DLL may act as its own wrapper.

The result is that any Windows protocol (simulation system) implementation will be able to load and execute any component.

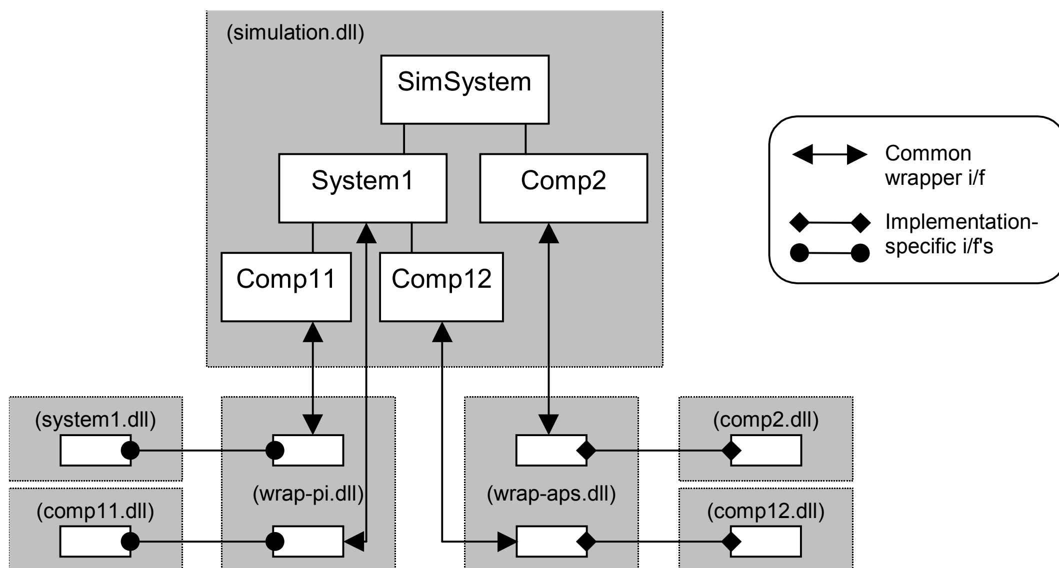


Figure 10.1. Component DLLs and their wrappers interacting with the protocol.

In the above diagram, the wrap-pi.dll shows a component wrapper implemented by CSIRO-PI and the wrap-aps.dll show a wrapper implemented by APSRU. Each of these wrappers will export the same functions so that the simulation.dll binary has a consistent interface to any functioning simulation components. The interfaces between the wrapper DLL's and the component binary's can be of a proprietary nature.

Providing communications between the simulation dll and the logic components.

A callback function is provided in the simulation dll for messages sent to it from any logic dll (or wrapper class). This callback follows this definition:

typedef _stdcall void (MCB)(const unsigned int *compInst, TMsgHeader *message);

where *compInst* points to the component instance in the simulation dll and *message* points to the message header being sent.

The component wrapper DLL's must export the following routines:

Routine	Description
`__stdcall void createInstance( const char *szLogicDLL, const unsigned int *lCompID, const unsigned int *lParentCompID, unsigned int *lInstanceID, const unsigned int *compInstance, MCB *messageCallback);`	Creates an instance of a component. `szLogicDLL` is the name of the DLL containing the component logic (input); `lCompID` is the instance's registration ID (input); `lOwnerID` is the registration ID of the system that manages the component (input); `lInstanceID` is a unique identifer used in later calls to the interface (output). Pointer to the instance of the logic dll (or wrapper class). compInstance pointer to the instance of the component within the system making this call. messageCallback function pointer to the entry into the system for messages sent back into the system.
`__stdcall void deleteInstance( int* lInstanceID )`	Deletes the component instance denoted by `lInstanceID`.
`__stdcall void messageToLogic( unsigned int *lInstanceID, TMsgHeader* message, bool* bProcessed )`	Passes a message to the component instance denoted by `lInstanceID`. `bProcessed` returns TRUE i.f.f. the component logic has carried out all processing necessary for the message.

- Memory management: The receiver of any message across the simulation dll – wrapper interface must take a copy of the message. The sender keeps ownership of the message and can delete it after it has been sent.
- The definition of `TMsgHeader` follows the layout of the message header in section 9.1.
- Because multiple simulations may be running concurrently and using the same wrapper DLL, the component registration ID (which is only unique within a single simulation) is insufficient as a unique instance identifier.
- Knowing whether or not a message was processed by the component logic is useful in the implementation of services.

10.1.3. Distributing the Simulation over more than one machine.

To enable the sharing of the simulation components over more than one machine address space it would be possible to build a component binary which acts as an interface to another component residing on another machine.

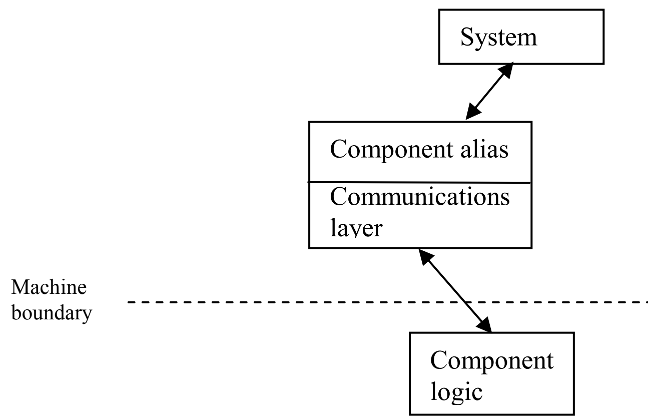


Figure 10.1 *Logical view of a distributed system*

10.2 Note on system implementations

In the majority of cases, a system that receives a message responds by routing it toward its destination component. If the receiving system manages the destination component, this is straightforward. If not, the receiving system will require some means of determining whether the message should be routed
(a) "up" to the system that manages the current system's system; or
(b) "down" to the system of one of the sub-systems of the system managed by the system.

Note: The PI implementation ensures that a system knows which components belong to it at any level. Because the components are all registered with the Simulation via calls routed through parent systems, it is not difficult for a System to store the ID's of any owned components.

11. References

Beek J & Frissel MJ (1973). *Simulation of nitrogen behaviour in soils.* PUDOC, Wageningen.

Brouwer R & de Wit CT (1968). A simulation model of plant growth with special attention to root growth and its consequences. *Proceedings of the 15th Easter School of Agriculture Science,* 224-242.

Christian KR, Freer M, Davidson JL, Donnelly JR & Armstrong JS (1978). *Simulation of grazing systems.* PUDOC, Wageningen.

Donnelly JR, Moore AD & Freer M (1997). GRAZPLAN: decision support systems for Australian grazing enterprises. I. Overview of the GRAZPLAN project, and a description of the MetAccess and LambAlive DSS. *Agricultural Systems* **54**, 57-76.

Freer M, Davidson JL, Armstrong JS & Donnelly JR (1970). Simulation of summer grazing. *Proceedings of the XI International Grassland Congress*, 913-917.

McCown RL, Hammer GL, Hargreaves JNG, Holzworth DP & Freebairn DM (1996). APSIM: a novel software system for model development, model testing, and simulation in agricultural systems research. *Agricultural Systems* **50**, 255–71.

O'Neill RV, De Angelis DL, Waide JB & Allen TFH (1986). *A Hierarchical Concept of Ecosystems*. Princeton University Press, Princeton NJ.

Appendix: Dimensions and SI Units

There is a difference between **dimensions** and **units**. A dimension is a measure of a physical variable (without numerical values), while a unit is a way to assign a number or measurement to that dimension. For example, length is a dimension, but it is measured in units of feet (ft) or metres (m).

Primary dimensions are defined as independent or fundamental dimensions, from which other dimensions can be obtained. There are seven primary dimensions:

Primary Dimension	Symbol	SI unit
Mass	M	g (gram)
Length	L	m (metre)
Time	T	s (second)
Temperature	θ	K (Kelvin)
Electric current	i	A (ampere)
Amount of light	I	cd (candela)
Amount of matter	N	mol (mole)

Other dimensions and units are **derived** from the primary dimensions and units as products and powers. The SI system defines a further 22 derived units:

Derived quantity	Name	Symbol	Definition using primary units
Plane angle	radian	rad	$m\ m^{-1} = 1$
Solid angle	steradian	sr	$m^2\ m^{-2} = 1$
Frequency	hertz	Hz	s^{-1}
Force	newton	N	$m\ kg\ s^{-2}$
Pressure, stress	pascal	Pa	$m^{-1}\ kg\ s^{-2}$
Energy, work, quantity of heat	joule	J	$m^2\ kg\ s^{-2}$
Power, radiant flux	watt	W	$m^2\ kg\ s^{-3}$
Electric charge, quantity of electricity	coulomb	C	s A
Electric potential difference, electromotive force	volt	V	$m^2\ kg\ s^{-3}\ A^{-1}$
Capacitance	farad	F	$m^{-2}\ kg^{-1}\ s^4\ A^2$
Electric resistance	ohm	Ω	$m^2\ kg\ s^{-3}\ A^{-2}$
Electric conductance	siemens	S	$m^{-2}\ kg^{-1}\ s^3\ A^2$
Magnetic flux	weber	Wb	$m^2\ kg\ s^{-2}\ A^{-1}$
Magnetic flux density	tesla	T	$kg\ s^{-2}\ A^{-1}$
Inductance	henry	H	$m^2\ kg\ s^{-2}\ A^{-2}$
Celsius temperature	degree Celsius	ºC	K
Luminous flux	lumen	lm	$m^2\ m^{-2}\ cd = cd$
Illuminance	lux	lx	$m^2\ m^{-4}\ cd = m^{-2}\ cd$
Activity (of a radionuclide)	becquerel	Bq	s^{-1}
Absorbed dose, specific energy (imparted), kerma	gray	Gy	$m^2\ s^{-2}$
Dose equivalent (d)	sievert	Sv	$m^2\ s^{-2}$
Catalytic activity	katal	kat	s^{-1} mol

The above table has been taken from the US National Institute of Standards and Technology website: http://www.physics.nist.gov/cuu/Units/units.html