COPASI C++ API Documentation

Version 4.8 (Build 35)

COLLABORATORS			
	TITLE : COPASI C++ API Documentation		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	COPASI Development Team	December 23, 2011	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

List of Tables

Chapter 1

C++

1.1 Introduction

This document tries to give a superficial overview of the COPASI backend API. This documentation is in no way meant to give a complete reference to the API. It is only intended to give the programmer that wants to use the library or the language bindings an overview of the classes and some examples on how certain things can be done. The language bindings expose parts of the underlying C++ API and so far no attempt has been made to simplify the API to better fit the target language or to make it easier for the programmer. All there is so far is a way to use COPASIs classes and methods from several languages.

1.2 Initialization of the library

Before the COPASI backend library can be used in C++, it has to be initialized. Initialization should be done as follows:

```
// initializes the root container
// argc and argv are the parameters that the main rountine gets
// alternatively, 0 and NULL may be passed as argc and argv.
CCopasiRootContainer::init(argc, argv);
```

It is best to do the initialization in a try...catch block.

In order to destroy the root container, e.g. at the end of the program, there is the static method CCopasiRootContainer::destroy.

Normally, init should only be called once at the beginning of a program and destroy at the end of the program. If needed, the current root container can also be deleted in the middle of a running program and you first have to delete the old root container if you want to initialize a new one with init.

In order to be more portable, the C++ backend of COPASI makes extensive use of typedefs for several datatypes. All those typedefs can be found in the copasi.h header file which has to be included by all programs that want to use the COPASI backend library in a C++ program.

And last, but not least, the statement "#define COPASI_MAIN" should occur at the very beginning (before copasi.h is included) of your source file that contains the main routine, otherwise, you will not be able to link against the COPASI library.

1.3 CCopasiRootContainer

In the COPASI backend, currently everything is located in an instance of CCopasiRootContainer. There is only one root container.

The CCopasiRootContainer class has a number of other useful methods that allow the user to aquire handles to the function database or the list of datamodels.

The COPASI backend only contains a single function database which contains all predefined and user defined function definitions. This function database is shared by all datamodels. In order to get a pointer to this function database, you can use the static method getFunctionList in CCopasiContainer. The method takes no arguments and returns a pointer to the function database which is an instance of the class CFunctionDB.

The other important data structure that is located in the root container is the list of datamodels. Earlier versions of the COPASI backend could only handle on datamodel at a time, but newer versions now allow the user to work with several datamodels at the same time.

After the root container has been initialized by calling init, it does not contain any datamodel yet. In order to create a new datamodel and add it to the root container, you have to call the static method addDatamodel in CCopasiRootContainer. The method takes no arguments and returns a pointer to the newly created instance of CCopasiDataModel. Since the datamodel is owned by the root container, it will be deleted when the root container is deleted.

In order to get the list of all datamodels, there is a static method in CCopasiContainer called getDatamodelList.

1.4 CCopasiDataModel

CCopasiDataModel is the class the contains the actual model, all tasks and all output definitions (report and plots).

In order to create a new model, you can call the static method addDatamodel in CCopasiRootContainer which will return the newly created datamodel. Or if the datamodel already has been created, you can get it via its index in the list of datamodels which you can get from the root container by calling getDatamodelList.

getDataModelList returns a list of the datamodels that have been created. This list has a method size to query the number of datamodels it contains.

If you want to delete a datamodel before the end of your program, CCopasiVector has a method remove which you can call with an index. This call will remove the object at the given index from the vector and free the memory that was allocated for it.

Caution Currently, it is not a good idea to change the name of a datamodel instance (e.g. by calling setObjectName). The backend relies on the datamodel to have a certain name and if that name changes, the backend will stop working for that datamodel. We hope to fix this problem in upcoming releases of COPASI.

CCopasiDataModel contains the actual model which can be obtained with the method *getModel()*, the defined tasks which can be obtained with the method *getTaskList()*, the reports which are obtained via the method *getReportDefinitionList()* and the function database which can be obtained with the *getFunctionList()*. In addition to these methods *CCopasiDataModel* provides some methods for loading COPASI files or importing SBML files, for saving COPASI files or exporting SBML files, for creating new models and for adding tasks to the task list.

1.5 Working with the model

1.5.1 COPASIs Model Concept

The CCopasiDataModel class contains the actual model as well as the tasks and all the output definitions (plots and reports). Newer versions of COPASI allow the user to work with several instances of the CCopasiDataModel class at the same time.

New instances can be create by the static method addDatamodel in CCopasiRootContainer and a list of the already created datamodels can be acquired with the static method getDatamodelList also from CCopasiRootContainer.

CCopasiDataModel contains the actual model which can be obtained with the method *getModel()*, the defined tasks which can be obtained with the method *getTaskList()*, the reports which are obtained via the method *getReportDefinitionList()* and the function database which can be obtained with the *getFunctionList()*. In addition to these methods *CCopasiDataModel* provides some methods for loading COPASI files or importing SBML files, for saving COPASI files or exporting SBML files, for creating new models and for adding tasks to the task list.

1.5.2 Model Elements

The class in COPASI that contains the actual model data is *CModel*. The model can be obtained from the instance of *CCopasiDataModel* via the *getModel()* method. To create a new empty model, *CCopasiDataModel* provides the method *newModel()*. There is also the method *loadModel(std::string filename)* which loads a COPASI model from the given filename, the method *importSBML(std::string filename)* which imports an SBML model from the file with the given name and last but not least the method *importSBMLFromString(std::string sbmlModelText)* which imports an SBML model from the given string. The string must be a valid SBML model string. All three methods return a boolean value which is *true* if the method succeeded and *false* otherwise.

Once the model has been obtained, it can be manipulated in many ways. New model elements like compartments, metabolites, reactions, events and/or global parameters can be created, modified or removed. The model also has some methods to manipulate the time-, volume- and substance units of the model. Unlike SBML, models in COPASI only have global units and the units of all elements of the model are made up of these global units. Units on individual elements are not supported.

To find out how many compartments, metabolites, reactions, events or global parameters (model values) a model has, *CModel* has the following methods.

```
// return the number of compartments in the model
C_INT32 getCompartments().size();
// return the number of metabolites in the model
C_INT32 getMetabolites().size();
// return the number of global parameters in the model
C_INT32 getModelValues().size();
// return the number of reactions in the model
C_INT32 getReactions().size();
// return the number of events in the model
C_INT32 getEvents().size();
```

To obtain a specific model element there are the following methods:

```
CCompartment* getCompartments()[const unsigned C_INT32& index];
CMetab* getMetabolites()[const unsigned C_INT32& index];
CModelValue* getModelValues()[const unsigned C_INT32& index];
CReaction* getReactions()[const unsigned C_INT32& index];
CEvent* getEvents()[const unsigned C_INT32& index];
```

New model elements can be created with the following methods:

```
/**
 * Creates a new compartment with the given name
 * and an initial volume of 1.0 volume units if no
 * volume is given. If a volume is given, the initial
 * volume will be set to that value.
 * The newly created compartment is added to the model
 * and the method also returns a pointer to the new instance.
 * If the method fails, NULL is returned.
 */
CCompartment* createCompartment(const std::string& name, const C_FLOAT64& volume = 1.0);

/**
 * Creates a new metabolite with the given name.
 * The metabolite is added to the compartment that is specified
 * as the second argument.
 * If a value for the initial concentration is given, the initial
 * concentration is set to that value, otherwise the
 * initial concentration is set to 1.0 substance units/volume units.
 * The newly created metabolite is added to the model and
 * the method also returns the new instance.
 * If the method fails, NULL is returned.
 */
CMetab* createMetabolite(const std::string& name,
```

```
                          const std::string& compartment
                          const C_FLOAT64& iconc = 1.0);


/**
 * Creates a new model value with the given name
 * and initial value.
 * If no initial value is given, it is assumed to be 0.0.
 * The newly created model value is added to the model
 * and the method also returns the new instance.
 * If the method fails, NULL is returned.
 */
CModelValue* createModelValue(const std::string name, const C_FLOAT64& value = 0.0);

/**
 * Creates a new reaction with the given name.
 * The newly created reaction is added to the model and
 * the method also returns the reaction instance.
 * If the method fails, NULL is returned.
 */
CReaction* createReaction(const std::string& name);

/**
 * Creates a new event with the given name.
 * The newly created event is added to the model and
 * the method also returns the event instance.
 * If the method fails, NULL is returned.
 */
CEvent* createEvent(const std::string& name);
```

Naturally model elements can also be removed via their keys. Each object in COPASI which is derived from *CCopasiObject* gets a unique id when it is created which in COPASI is called key. To get the key for such an object, *CCopasiObject* provides the method *getKey(). CModel*, *CCompartment*, *CMetab*, *CModelValue* and *CReaction* are all derived from *CCopasiObject* and you can therefore get the key for any of them via the *getKey()* method which returns the key in the form of a string.

The methods to delete elements from the model are:

```
bool removeCompartment(const std::string& key);
bool removeMetabolite(const std::string& key);
bool removeModelValue(const std::string& key);
bool removeReaction(const std::string& key);
bool removeEvent(const std::string& key);
```

These methods also delete all dependent objects. So if a compartment is deleted, all metabolites in that compartment are also deleted and if those metabolites were part of reaction those reactions are deleted as well.

For example, to delete the third metabolite in the model you could write the following code (provided that there are at least three metabolites in the model):

```
CModel* pModel=pDatamodel->getModel();
CMetab* pMetab=pModel->getMetabolite(3);
std::string key=pMetab->getKey();
pModel->removeMetabolite(key);
```

Other methods that might be of interest in the CModel class are e.g. the methods to calculate the jacobian matrix and the reduced jacobian matrix of the model (see example 8).

```
/**
 * Calculates the jacobian of the full model for the current state
 * and stores it in the provided matrix. applyInitialValues()
 * needs to be called before.
 * @param CMatrix< C_FLOAT64 > & jacobian
```

```
 * @param const C_FLOAT64 & derivationFactor,
 * @param const C_FLOAT64 & resolution
 */
void calculateJacobian(CMatrix< C_FLOAT64 > & jacobian,
                       const C_FLOAT64 & derivationFactor,
                       const C_FLOAT64 & resolution);

/**
 * Calculates the Jacobian of the reduced model for the current
 * state and stores it in the provided matrix. applyInitialValues()
 * needs to be called before.
 * @param const C_FLOAT64 & derivationFactor,
 * @param const C_FLOAT64 & resolution
 * @param CMatrix< C_FLOAT64 > & jacobianX
 */
void calculateJacobianX(CMatrix< C_FLOAT64 > & jacobianX,
                        const C_FLOAT64 & derivationFactor,
                        const C_FLOAT64 & resolution);

/**
 * initialize all values of the model with their initial values
 */
void applyInitialValues();
```

1.5.2.1 CModelEntity

In COPASI the classes for the model itself, for compartments, metabolites and global parameters are derived from *CModelEntity* and inherit all its methods. Most of the time, if you manipulate any of those classes, it will be done through methods from *CModelEntity*. *CModelEntity* provides the following methods:

```
/**
 * Gets the status of the entity.
 */
const CModelEntity::Status& getStatus() const;

/**
 * sets the status of an entity.
 */
void setStatus(const CModelEntity::Status& status);

/**
 * gets the transient value of an entity.
 */
const C_FLOAT64& getValue() const;

/**
 * Returns the initial value of an entity.
 */
const C_FLOAT64& getInitialValue() const;

/**
 * Sets the initial value of an entity.
 */
void setInitialValue(const C_FLOAT64& initialValue);

/**
 * Returns the SBML id of an entity. If none has been set,
 * an empty string is returned.
 */
const std::string& getSBMLId() const;
```

```
/**
 * Sets the SBML id on an entity.
 * This is either done on importing an SBML file,
 * or on exporting a model.
 * Don't mess with this since it might break further
 * exports to SBML.
 */
void setSBMLId(const std::string& id);

/**
 * If the entity is determined by an assignment or
 * an ODE, this method will return the mathematical
 * expression for the assignment or the ODE.
 */
std::string getExpression() const;

/**
 * With this method, the mathematical expression
 * for entities that are determined by an assignment
 * or an ODE can be set.
 * You might also have to set the status of the entity
 * to indicate that it is now determined by an ODE
 * or an assignment. (see setStatus())
 */
bool setExpression(cosnt std::string& expression);

/**
 * Returns the mathematical expression for any initial
 * assignment that might have been set on the entity.
 * This is independent of the status of the entity.
 */
std::string getInitialExpression() const;

/**
 * With this method, the mathematical expression
 * for the initial assignment of an entity can be set.
 */
bool setInitialExpression(const std::string& expression);
```

The *setStatus(int status)* and *getStatus()* methods set and return the status of an entity respectively. The status of an entity determines how it is calculated. E.g. an entity that has a status of FIXED is not changed during a time course simulation, whereas an entity that has a status value of ASSIGNMENT is determined by an assignment expression. For a full list of valid status definitions see table model entity status.

state	description
FIXED	entity is fixed
ASSIGNMENT	entity is determined by an assignment
REACTIONS	entity is determined by reactions (only applicable to metabolites)
ODE	entity is determined by an ode
TIME	this is the only state the model itself can have

Table 1.1: model entity status

1.5.2.2 compartment

The *CCompartment* class is derived from *CModelEntity* and inherits all its methods. Compartments can have the status FIXED, ASSIGNMENT or ODE. And you can set the corresponding mathematical expressions for the assignment or ODE through the

methods provided by *CModelEntity*. You can also either set an initial value or a mathematical expression that determines the initial value of the compartment. In addition to the methods provided by *CModelEntity*, *CCompartment* provides some methods to handle the metabolites in a compartment.

```
/**
 * Returns a CCopasiVectorNS<CMetab> object which contains
 * all metabolites that are contained in the compartment.
 * CCopasiVectorNS has a size method to determine the
 * number of metabolites and a get method that takes an
 * index and returns the metabolite at that index.
 * If you need to remove a metabolite from a compartment,
 * the vector also has a method remove which takes the pointer
 * to the object that needs to be removed as the argument.
 */
CCopasiVectorNS<CMetab>& getMetabolites();
const CCopasiVectorNS<CMetab>& getMetabolites() const;

/**
 * Adds the given metabolite to the compartment.
 * If the operation succeeded, true is returned, else false.
 */
bool addMetabolite(CMetab* pMetabolite);
```

1.5.2.3 metabolites

Just like the compartment class, the class for metabolites, *CMetab* is derived from *CModelEntity* and inherits all its methods. Metabolites can have the status FIXED, ASSIGNMENT, ODE or REACTIONS. And you can set the corresponding mathematical expressions for the assignment or ODE through the methods provided by *CModelEntity*. You can also either set an initial value or a mathematical expression that determines the initial value of the metabolite. Using the methods from *CModelEntity* to set or get values from a metabolite means that you get or set the particle number of the metabolite. If you want to work with concentrations rather than particle numbers, *CMetab* provides the methods *getConcentration()*, *getInitialConcentration()* and *setInitialConcentration(double c)* to get the concentration, initial concentration or set the initial concentration respectively. The class also provides the method *getCompartment()* which returns the compartment of the metabolite.

1.5.2.4 model values (global parameters)

In COPASI global parameters are called model values and the corresponding class is *CModelValue* which is also derived from *CModelEntity*. A global parameter can also be determined by an assignment or an ODE or it can be fixed. Global parameters can also have initial assignments.

1.5.2.5 reactions

The class for reaction is called *CReaction*. It contains information on the metabolites taking part in the reaction as substrates, products or modifiers and about the kinetic law.

```
/**
 * Returns the current flux of the reaction.
 */
const C_FLOAT64& getFlux();

/**
 * Returns the current flux of the reaction in terms
 * of particle numbers rather than concentrations
 */
const C_FLOAT64& getParticleFlux();

/**
 * If the reaction is reversible, this returns true, else false.
```

```
 */
bool isReversible();

/**
 * Sets whether the reaction is reversible or not.
 */
void setReversible(bool reversible);

/**
 * Returns the number of compartments that are involved
 * in this reaction.
 */
unsigned C_INT32 getCompartmentNumber() const;

/**
 * Returns the SBML id of the reaction if one has been set,
 * otherwise an empty string is returned.
 */
const std::string& getSBMLId() const;

/**
 * Sets the SBML id of the reaction.
 * This is done during import of an SBML model, or
 * during the first export of an SBML model.
 * If you change the SBML id of any entity, further
 * exports to SBML might fail.
 */
void setSBMLId(const std::string& id);

/**
 * Returns the chemical equation object of the reaction.
 */
const CChemEq& getChemEq() const;
CChemEq& getChemEq();

/**
 * Returns the function for the kinetic law used in the reaction.
 */
CFunction* getFunction();

/**
 * Sets the kinetic law of the reaction via a functions name.
 * A function with that name must exist in the function database.
 */
bool setFunction(const std::string& functionName);
```

The *getFunction* method returns the function that is used as the kinetic law. In COPASI all kinetic laws are determined by function calls. Using an expression as a kinetic law is not supported. All kinetic law expressions that are imported from SBML files are first converted to function definitions with a call to that function as the kinetic law. When those models are reexported, all expressions are exported as function calls. There are exceptions to this rule however. If COPASI determines that a given rate law is a Mass Action rate law, no function definition is created but the Mass Action function build into COPASI is used as the kinetic law. On reexport to SBML, the kinetic law is converted to a Mass Action expression again. In the future, similar procedures might be used for other function types as for example constant flux.

The chemical equation of a reaction is stored in a *CChemEq* object which can be acquired from the reaction with a call to *getChemEq*. The chemical equation object has several methods to query the properties of the chemical equation:

```
/**
 * Returns true if the chemical reaction is reversible.
 */
const bool& getReversibility() const;
```

```
/**
 * Sets whether the given chemical equation is to be reversible.
 */
void setReversibility(const bool& revers);

/**
 * Adds a new metabolite to the chemical equation.
 * key is the key of the metabolite
 * (keys are the ids of COPASI objects)
 * mult is the stoichiometric coefficient
 * role is the role of the metabolite and it can be either
 * CChemEq::PRODUCT, CChemEq::SUBSTRATE, CChemEq::MODIFIER
 * or CChemEq::NOROLE.
bool addMetabolite(const std::string& key, const C_FLOAT64 multiplicity, const  ↩
    MetaboliteRole& role);

/**
 * Gets a container with all the substrates of the
 * chemical equation.
 * The size of the container can be queried with
 * its size method,
 * the element with index i can be retrieved with the
 * get method which takes the index as the only argument.
 */
const CCopasiVector<CChemEqElement>amp; getSubstrates() const;

/**
 * Gets a container with all the products of the
 * chemical equation.
 * The size of the container can be queried with
 * its size method,
 * the element with index i can be retrieved with the
 * get method which takes the index as the only argument.
 */
const CCopasiVector<CChemEqElement>amp; getProducts() cosnt;

/**
 * Gets a container with all the modifiers of the
 * chemical equation.
 * The size of the container can be queried with its
 * size method,
 * the element with index i can be retrieved with the
 * get method which takes the index as the only argument.
 */
const CCopasiVector<CChemEqElement>amp; getModifiers() const;

/**
 * Returns the number of compartments that are
 * involved in this reaction.
 * If all metabolites that are involved in the
 * reaction are in the same compartment 1 is returned.
 */
unsigned C_INT32 getCompartmentNumber() const;

/**
 * Returns the largest compartment involved in the reaction.
 * This method should only be called after all transient values have been set,
 * otherwise an exception can occur.
 * Maybe call applyInitialValues on the model before using this method.
 */
const CCompartment* getLargestCompartment() const;
```

The individual entities of the chemical reaction (substrates, products and modifiers) are represented by *CChemEqElement* instances. A *CChemEqElement* object stores the stoichiometric coefficient of the reaction entity and the key of the associated *CMetab* object. Those properties can be queried and manipulated with the following methods:

```
/**
 * Returns the stoichiometric coefficient of the element.
 */
C_FLOAT64 getMultiplicity() const;

/**
 * Sets the stoichiometric coefficient of the element.
 */
void setMultiplicity(const C_FLOAT64 multiplicity);

/**
 * Returns the associated CMetab instance.
 */
const CMetab* getMetabolite() const;

/**
 * Returns the key of the associated CMetab instance.
 */
const std::string& getMetaboliteKey() const;

/**
 * Sets the associated CMetab instance via its key.
 */
void setMetabolite(const std::string& key);
```

1.5.2.6 events

The class for events is called *CEvent*. It contains information on the condition that determines when an event is triggered, the assignments that are executed when the event is triggered and an optional delay expression that determines how long after the actual triggering of the event, the assignments are applied.

```
/**
 * Set the order in which the event is executed for simultaneous events.
 * @param const unsigned C_INT32 & order
 * const bool & correctOther = true
 */
void setOrder(const unsigned C_INT32 & order, const bool & correctOther = true);

/**
 * Retrieve the order in which the event is executed for simultaneous events
 */
const unsigned C_INT32 & getOrder() const;

/**
 * Sets the SBMLId of the event.
 * @param const std::string & id
 */
void setSBMLId(const std::string & id);

/**
 * Returns a reference to the SBML Id of the event.
 */
const std::string& getSBMLId() const;

/**
 * Set whether the calculation or the assignments shall be delayed
 */
```

```
void setDelayAssignment(const bool & delayCalculation);

/**
 * Retrieve whether to delay the calculation of the assignments.
 * @return const bool & delayCalculation
 */
const bool & getDelayAssignment() const;

/**
 * Set the expression of trigger from a string. The return value indicates if
 * parsing the expression was successful.
 * @param const std::string & expression
 * @return bool success
 */
bool setTriggerExpression(const std::string & expression);

/**
 * Set the expression of trigger from a CExpression instance.
 * The pointer is owned by the event after it has been set.
 * @param CExpression* pExpression
 */
void setTriggerExpressionPtr(CExpression * pExpression);

/**
 * Retrieve the expression of trigger as a string.
 * @return std::string expression
 */
std::string getTriggerExpression() const;

/**
 * Retrieve the const pointer to the trigger expression.
 * @return const CExpression* pExpression
 */
const CExpression * getTriggerExpressionPtr() const;

/**
 * Retrieve the pointer to the trigger expression.
 * @return CExpression* pExpression
 */
CExpression * getTriggerExpressionPtr();

/**
 * Set the expression of delay from a string. The return value indicates if
 * parsing the expression was successful.
 * @param const std::string & expression
 * @return bool success
 */
bool setDelayExpression(const std::string & expression);

/**
 * Set the expression of delay from a CExpression instance pointer.
 * The pointer is owned by the event after it has been set.
 * @param CExpression* pExpression
 */
void setDelayExpressionPtr(CExpression* pExpression);

/**
 * Retrieve the expression of the delay expression as a string.
 * @return std::string expression
 */
std::string getDelayExpression() const;
```

```
/**
 * Retrieve the pointer to the expression of the delay.
 * @return CExpression* pExpression
 */
CExpression* getDelayExpressionPtr();

/**
 * Retrieve the const pointer to the expression of the delay.
 * @return const CExpression* pExpression
 */
const CExpression* getDelayExpressionPtr() const;

/**
 * Retrieve the assignments. This will return a const reference to the CCopasiVectorN which ↩
     holds the assignments for the event.
 * @return const CCopasiVectorN< CEventAssignment > & assignments
 */
const CCopasiVectorN< CEventAssignment > & getAssignments() const;

/**
 * Retrieve the assignments. This will return the reference to the CCopasiVectorN which  ↩
    holds all the assignments.
 * @return CCopasiVectorN< CEventAssignment > & assignments
 */
CCopasiVectorN< CEventAssignment > & getAssignments();

/**
 * Delete assignment with the given key. Please note this is not the target key.
 * @param const std::string & key
 */
void deleteAssignment(const std::string & key);
```

The individual assignments that have to be executed when an event fires are represented by the CEventAssignment class in COPASI. Each event assignment has a target which is the object that is changed and an expression that determines the value to which the target is set.

```
/**
 * Set the key of the target
 * @param const std::string & targetKey
 * @return bool success;
 */
bool setTargetKey(const std::string & targetKey);

/**
 * Retrieve the target key
 * @return const std::string & targetKey
 */
const std::string & getTargetKey() const;

/**
 * Retrieve a pointer to the target object.
 * @return const CCopasiObject * targetObject
 */
const CCopasiObject * getTargetObject() const;

/**
 * Set the expression from an infix string. The return value indicates if
 * parsing the expression was successful.
 * @param const std::string & expression
 * @return bool success
 */
bool setExpression(const std::string & expression);
```

```
/**
 * Set the expression from an expression pointer. CEventAssignment takes ownership.
 * @param CExpression* pExpression
 */
void setExpressionPtr(CExpression * pExpression);

/**
 * Retrieve the expression as a string.
 * @return std::string expression
 */
std::string getExpression() const;

/**
 * Retrieve the pointer to the expression.
 * @return CExpression * pExpression
 */
const CExpression * getExpressionPtr() const;

/**
 * Retrieve the pointer to the expression.
 * @return CExpression * pExpression
 */
CExpression * getExpressionPtr();
```

1.5.3 The Function Database

The function database is the place where COPASI stores all function definitions, that is all builtin functions as well as all used defined functions. There is only one global function database and it is not associated with the current model. If the model is deleted, all functions that were used in the model stay in the function database.

The function database object is an instance of the class *CFunctionDB* and the global function database can be acquire from the root container:

```
CFunctionDB* funDB=CCopasiRootContainer::getFunctionList();
```

The *CFunctionDB* class has a number of methods to get, remove or add new function definitions.

```
/**
 * Finds the function with the given name.
 * If no function with this name can be found, NULL is returned.
 */
CEvaluationTree* findFunction(const std::string& functionName);

/**
 * Returns a container with all known function definitions.
 * The size of the container can be queries with the size
 * method and individual elements of type CEvaluationTree
 * can be acquire with the get method. The only argument to
 * the get method is the index of the desired function.
 */
CCopasiVectorN<CEvaluationTree<amp; loadedFunctions();

/**
 * Returns a container with all functions that are suitable
 * for the given number of substrates and products and the
 * given reversibility.
 * noSubstrates: desired number of substrates
 * noProducts: desired number of products
 * reversibility: whether the function should be suitable for
```

```
 * reversible or irreversible reactions.
 */
std::vector<CFunction*> suitableFunctions(const unsigned C_INT32 noSubstrates,
                                const unsigned C_INT32 noProducts, const TriLogic ↩
                                    reversibility);

/**
 * Removes the function definition with the given key.
 */
bool removeFunction(const std::string& key);

/**
 * Removes all functions from the function database.
 * You should only call this method if you do not have a model loaded
 * that uses the functions.
 */
void cleanup();

/**
 * Loads the builtin functions into the function database.
 * Calling this after calling cleanup will basically reinstantiate
 * the function database.
 */
void load();

/**
 * Adds the given function to the COPASI function database.
 * The second argument determines whether COPASI will take ownership
 * of the function or if ownership remains with the calling program.
 * If the second argument is specified as false, the programmer is
 * responsible for eventually deleting the function object.
 *
 * The return value indicates whether adding the function was successfull
 * or not.
 */
bool add(CFunction* pFunction, bool adopt);
```

Caution Since COPASI uses only a single function database, this database will keep growing as you load or import models. As one of our users noticed, this will eventually lead to a severe slowdown while importing SBML models. In order to get around this problem, you can restore the function database to it's initial state once in a while by calling cleanup and then load on the function database object. However, you should only do this while there is no model loaded that uses these functions. So it is best to first remove all instances of *CCopasiDataModel* (see removeDataModel from *CCopasiRootContainer*).

The individual function definitions are instances of the class *CEvaluationTree* or of subclasses of that class, e.g. *CFunction*. COPASI distinguishes between different types of function definitions:

CEvaluationTree::MassAction this is the type of the mass action kinetic function definition

CEvaluationTree::PreDefined this type is set on all builtin function definitions that are not mass action

CEvaluationTree::UserDefined this type is set on all user defined function definitions

The type of a *CEvaluationTree* object can be queried with the *getType* method. The formula that is associated with a function definition can be queried with the *getInfix* method. This method returns the equation as a ASCII string. In order to set the equation one can use the *setInfix* method. This method takes the formula as an ASCII string and returns *true* if the equation has been set successfully. If there was an error, the method returns *false*.

Besides the mass action function definition which is an instance if the class *CMassAction* (for which there are currently no wrapper functions) all function definitions are instances of *CFunction*. In addition to the method from its base class *CEvaluationTree*, *CFunction* has some additional methods to handle the function parameters and some other properties.

```
/**
 * If the reaction is reversible, the return value
 * TriTrue,
 * if the reaction is not reversible it is
 * TriFalse.
 * If the function is not meant to be used as a kinetic law,
 * the result might  also be TriUnspecified.
 */
const TriLogic& isReversible() const;

/**
 * Sets whether the function is meant to be used on reversible
 * reaction, irreversible reactions, or if it is a general function
 * not meant to be used as a kinetic law.
 * The allowed values are TriTrue,
 * TriFalse or TriUnspecified, respectively.
 */
void setReversible(const TriLogic& reversible);

/**
 * Returns a CFunctionParameters structure that holds
 * all parameters for the function definition.
 */
CFunctionParameters& getVariables();
const CFunctionParameters& getVariables() const;

/**
 * Returns the index of the variable with the given name.
 * If there is no function parameter with that name,
 * -1 is returned.
 */
unsigned C_INT32 getVariableIndex(const std::string& name) const;

/**
 * Returns true if the function definition is suitable for the
 * given number of substrates, products and the given reversibility.
 */
bool isSuitable(const unsigned C_INT32 noSubstrates, const unsigned C_INT32 noProducts,
                          const TriLogic reversible);
```

When the equation is set for a function definition, the parameters are parsed and added to the function definition.

1.5.4 Setting initial values on model entities

It is not obvious what consequences it has when the initial values of model entities are changed because changing the initial value of a model entity might influence other initial values, e.g. compartment volumes influence the concentration of the contained metabolites. Therefore, it is not enough to just set the initial values on model entities, but it is necessary to tell COPASI which objects have been changed so that COPASI can update dependent values. This update is executed with the *updateInitialValues* method of *CModel*. The method takes one argument which is a container that stores all initial values that have been changed in the model. The container is an instance of *ObjectStdVector* and store objects of type *CCopasiObject*. The objects stored there are the actual values that have been changed, not the model entities the values belong to, e.g. if the initial concentration of a metabolite has been changed, an object representing this initial concentration can be acquire by calling *getObject* on the model entity with a *CCopasiObjectName* as the argument that specifies what value to get. So assuming *container* is an instance of *ObjectStdVector* and *metab* is an instance of *CMetab* and the initial concentration of *metab* has been changed, we can add the object that corresponds to the initial concentration of *metab* with

```
pContainer->add(pMetab->getObject(CCopasiObjectName("Reference=InitialConcentration")));
```

The same procedure can be used for the initial volume of compartments, he initial value of parameters and the value of local reaction parameters. The respective strings to initialize the CCopasiObjectName instance are

```
"Reference=InitialVolume"
```

,

```
"Reference=InitialValue"
```

and

```
"Reference=Value"
```

. Once all changed values have been added to the container, *updateInitialValues* can by called on the instance of *CModel* with the container as the only argument.

1.6 Working with Tasks

1.6.1 The Task-Problem-Method Concept in COPASI

In COPASI calculations you can do on the current model are represented by so called tasks. The base class for all tasks is *CCopasiTask*. Each task contains a problem represented by a subclass of *CCopasiProblem* and a method represented by a subclass of *CCopasiMethod*. For a given task type, e.g. time course simulation, there is one task class, one problem class and one or more method classes. The problem of a task describes what is going to be done when the task is run and the method describes how it is done. Lets make this a little more clear by looking at the task class for running time course simulations on a model. The task class for this is *CTrajectoryTask* and is contains an instance of the problem class *CTrajectoryProblem*. The problem for the time course simulation stores the parameters for the simulation like start time, end time and number of steps. The method class for the time course task is a subclass of *CTrajectoryMethod* which itself is a subclass of *CCopasiMethod*. At the time of writing, there were six method classes derived from *CTrajectoryMethod*. Two of the method classes are for deterministic simulation, two are for stochastic simulation and two are for hybrid simulation. Each of the methods contains parameters that are specific to the method and which can be changed to influence the corresponding method.

All tasks are stored in *CCopasiDataModel*. *CCopasiDataModel* provides the method *getTaskList()* to get the list of tasks. The returned object is of the type *TaskVectorN* and provides a method called *size()* to find out how many tasks it contains and a method *get(C_INT32 index)* to retrieve the task with the given index. The object that is returned is of type *CCopasiObject*. Alternatively, *CCopasiDataModel* has a method called *getTask(C_INT32 index)* which returns the *CCopasiTask* object with the given index.

To get the problem and the method for a given task, *CCopasiTask* provides the methods *getProblem()* and *getMethod()* respectively and the objects returned are of type *CCopasiProblem* and *CCopasiMethod*. Normally there is no need to cast the task, problem or method objects to the most specific subclass since the base classes already provide all the needed functionality.

All that is needed to run any task is

1. get the correct task from the instance of *CCopasiDataModel*
2. if the task does not exist yet, create a new one and add it to the instance of *CCopasiDataModel*
3. change the parameters of the problem if the default values are not O.K.
4. change the method for the task if the one set is not the one that is wanted
5. change the parameters on the method if the default values are not O.K.

The procedure is (almost) the same no matter what the task might be, although some tasks might need some additional steps. What those additional steps are will be explained in the corresponding sections below.

The first thing is to get the correct task. For this the type of the task has to be known. The type of a task is an integer value and a list of all known types can be found in CCopasiTask.java. Some of those tasks are not available from Java yet. If we look at this list, we see that the type for the time course simulation task is CCopasiTask.timeCourse. So in order to find the time course task, we write a loop that checks all tasks that the CCopasiDataModel instance knows if one has the correct type. For each task type, there can be at most one task.

So to find the time course simulation task we could use the following code:

```
long i=0;
long iMax=pDataModel->getTaskList()->size();
CTrajectoryTask* pTask=NULL;
for(i=0;i<iMax;i++)
{
   if((*pDataModel->getTaskList())[i]->getType()==CCopasiTask::timeCourse)
   {
       pTask=(*pDataModel->getTaskList())[i];
       break;
   }
}
if(pTask==NULL)
{
   // create the task
   pTask=pDataModel->addTask(CCopasiTask::timeCourse);
}
```

The classes *CCopasiProblem* and *CCopasiMethod* are both derived from *CCopasiParameterGroup*, they inherit all the methods from *CCopasiParameterGroup* to set their parameters. There are two ways to get a parameter, the first method retrieves a parameter by its index, the second method gets the parameter via its name.

The getIndex method can be used to find if a parameter with a certain name exists within the given parameter group. If the parameter does not exist in the parameter group the size_t value C_INVALID_INDEX is returned by the C++ API.

```
CCopasiParameter* getParameter(const unsigned C_INT32& index);
const CCopasiParameter* getParameter(const unsigned C_INT32& index) const;
CCopasiParameter* getParameter(const std::string& name);
const CCopasiParameter* getParameter(const std::string& name) const;
size_t getIndex(const std::string& name) const;
```

*CCopasiParameterGroup*s methods *size()* can be used to find out how many parameters are part of the parameter group. Each instance of *CCopasiParameter* has a type that determines what kind of values are allowed for the parameter. E.g. it determines if the parameter represents an integer a float value or a string. Depending on the type of the parameter which can be acquire with the *type()* method, the parameters value has to be set with a specific method. Let us for example look at the parameter called "Duration" which is a parameter of CTrajectoryProblem and determines up to which time the simulation will be calculated. The parameter itself stores a double value. Assuming you already have the correct task, you can set this parameter as follows.

```
CTrajectoryProblem* pProblem=(CTrajectoryProblem*)pTask->getProblem();
CCopasiParameter* pParameter=pProblem->getParameter("Duration");
if(pParameter!=NULL && pParameter->getType()==CCopasiParameter::UDOUBLE)
{
    pParameter->setValue(100.3);
}
```

This would set the parameter so that running a simulation would simulate a time length of 100.3 time units.

The types that are allowed for a parameter are defined in CCopasiParameter.java. The defined types as of the time of writing are those listed in table parameter types.

Most of the time you will only have to deal with the numeric parameters of type DOUBLE, INT, UINT or BOOL.

Type	Description
DOUBLE	a double value
UDOUBLE	a positive double value
INT	an integer value (long)
UINT	a positive integer value (long)
BOOL	a boolean value (0 or 1)
GROUP	a parameter group (parameter groups can be nested
STRING	a String
CN	a COPASI common name (String)
KEY	a key (String)
FILE	a filename (String)

Table 1.2: parameter types

Next the method has to be set on the task. The method of a task can be set with the *setMethodType(int type)* method of *CCopasiTask* the argument to the method is an integer that specifies the type of the method. All methods known to COPASI have a specific type and all those types are defined in CCopasiMethod.java. Not all methods are applicable to all tasks, normally a task has a certain set of methods that can be used with the task. If the method type passed to *setMethodType* is not applicable to the task on which the method has been called, the method return a value of *false*, otherwise it returns *true*. To find out which method is currently set on a task, you can use a call to *getMethodType()* which will return an integer representing the method.

Which method can be used with which task will be described in the documentation for the individual tasks. One the correct method has been set, the method parameters can be changed which works exactly the same way as described for the parameter of the problem above, the only difference is that the methods are used in an instance of *CCopasiMethod* which can be obtained from the task with a call to *getMethod()*.

Once everything has been set, the task can be executed by calling the *process(boolean useInitialValues)* method. The boolean argument to the method determines if the task will use the current values of all model elements to run or if all values will be reset the the elements initial values. Using the current values rather than the initial values might be useful if a simulation has already been run, but one wants to extend this simulation to get more data points. When a task is run, the results are either stored in a report if one was defined, and/or in memory. How reports can be defined and how the results for a specific task can be obtained will be explained in the documentation for the specific tasks below.

1.6.2 Running Time Course Simulations

The procedure for running a time course simulation has more or less already been described above. What remains to be explained is what parameters can be used on *CTrajectoryProblem* (table trajectory problem parameters}) and which methods can be used with *CTrajectoryTask* (table trajectory methods).

Name	Type	Description
StepNumber	UINT	number of time steps that are calculated
StepSize	DOUBLE	size of each time step
Duration	DOUBLE	length of the simulation
TimeSeriesRequested	BOOL	determines if the result is to be stored in memory
OutputStartTime	DOUBLE	the time at which COPASI begins to produce output

Table 1.3: trajectory problem parameters

Here a complete example on how it could be done:

```
long i=0;
long iMax=pDataModel->getTaskList()->size();
```

Type	Description
deterministic	simulation with LSODA
LSODAR	simulation with LSODAR
stochastic	simulation with the next reaction method
hybrid	hybrid with next reaction method and runge kutta (4th order)
hybridLSODA	hybrid with next reaction method and LSODA
tauLeap	stochastic simulation with the tauLeap method

Table 1.4: trajectory method types

```
CTrajectoryTask* pTask=NULL;
for(i=0;i<iMax;i++)
{
   if((*pDataModel->getTasks())[i].getType()==CCopasiTask::timeCourse)
   {
       pTask=(*pDataModel->getTasks())[i];
       break;
   }
}
if(pTask==NULL)
{
   // create the task
   pTask=pDataModel->addTask(CCopasiTask::timeCourse);
}
if(pTask!=NULL)
{
  CTrajectoryProblem* pProblem=(CTrajectoryProblem*)pTask->getProblem();
  CCopasiParameter* pParameter=pProblem->getParameter("Duration");
  if(pParameter!=NULL && pParameter->getType()==CCopasiParameter::UDOUBLE)
  {
    pParameter->setValue(100.3);
  }
  if(pTask->setMethodType(CCopasiMethod::LSODAR))
  {
    // maybe change some method parameters
    pTask->initialize(CCopasiTask::OUTPUT_COMPLETE, pDataModel, NULL);
    pTask->process(true);
    pTask->restore();

  }
  else
  {
    // error handling
  }
}
```

After running the time course simulation the results are stored in memory if the "TimeSeriesRequested" parameter was set to *true*. In order to get the results from the simulation *CTrajectoryTask* provides the method *getTimeSeries()* which returns an instance of *CTimeSeries*. *CTimeSeries* itself provides some methods to read out the results.

```
/**
 * Returns the number of steps in the time series.
 * This should be the same number as the "StepNumber"
 * parameter of the CTrajectoryProblem instance plus
 * one for the initial state.
 */
const unsigned C_INT32& getRecordedSteps() const;

/**
```

```
 * Returns the number of variables in each row of
 * the result. Which is the same as the number of
 * columns. This includes the column for time.
 */
const unsigned C_INT32& getNumVariables() const;

/**
 * Gets the value for a specific row and column of
 * the result. The rows are the individual time steps
 * and the columns are the variables.
 * Indices begin with 0 and the first column (var=0) is
 * the time.
 * If the column represents a metabolite, this method
 * returns the particle number.
 */
const C_FLOAT64& getData(const unsigned C_INt32& step, const unsigned C_INT32& var) const;

/**
 * Gets the value for a specific row and column of
 * the result. The rows are the individual time steps
 * and the columns are the variables.
 * Indices begin with 0 and the first column (var=0)
 * is the time.
 * If the column represents a metabolite, this method
 * returns the concentration.
 * For all other columns, the value is the same as the
 * one you get with getData.
 */
const C_FLOAT64& getConcentrationData(const unsigned C_INT32& step, const unsigned C_INT32& ↩
    var) const;

/**
 * Get the key (id) of the element represented by the
 * column with index var.
 */
const std::string& getKey(const unsigned C_INT32& var) const;

/**
 * Get the SBML id of the element represented by the column
 * with index var.
 * Objects only have SBML ids if the model was imported from
 * an SBML file, or has already been exported to an SBML file.
 */
const std::string& getSBMLId(const unsigned C_INT32& var);
```

1.6.3 Performing Steady State Calculations

Performing a steady state calculation is very similar to running a time course simulation. There is a task instance (*CSteadyStateTask*) that contains a problem instance (*CSteadyStateProblem*) and a method instance (*CNewtonMethod*). At the time this was written, there is only one method for the steady state calculation, so we have to deal with a fixed set of parameters that can be set on the problem and the method. The steady state problem has two parameters. One (*JacobianRequested*) determines if the Jacobian matrix is to be calculated and the other (*StabilityAnalysisRequested*) determines if a stability analysis is to be done. Both values are boolean values and per default they are set to *true*.

The *CNewtonMethod* has the following parameters:

Resolution Resolution to determine if a steady state has been found (Double)

Derivation Factor Used to determine the step size used for numerical differentiation (Double)

Use Newton Use the Newton method to find steady states (Boolean)

Use Integration Use integration to find steady states (Boolean)

Use Back Integration Use backwards integration to find steady states (Boolean)

Accept Negative Concentrations Accept a steady state with negative concentrations (Boolean)

Iteration Limit Maximum number of iterations for the newton method (positive Integer)

For a detailed description of all these parameters please have a look in the COPASI manual in the section for the steady state method. To run the calculation, the *process* method of the task has to be used. The method takes two arguments. The first argument is a *boolean* value that determines whether the task is started with the current model values or with the initial values of all model entities. The second argument is an int value that can be either *CCopasiTask.NO_OUTPUT* or *CCopasiTask.OUTPUT_COMPLETE* which determined if output is generated from the task or not. This is important if reports have been defined for the task (see section Creating Reports).

After the task has finished, the results can be queried via several methods provided by the *CSteadyStateTask* instance.

```
/**
 * Returns an integer value. The value can either be
 * CSteadyStateMethod::notFound, CSteadyStateMethod::found,
 * CSteadyStateMethod::foundEquilibirium or
 * CSteadyStateMethod::foundNegative depending on whether
 * a steady state was found and what properties the
 * found steady state has.
 */
const CSteadyStateMethod::ReturnCode& getResult() const;

/**
 * If a steady state was found, the corresponding state
 * of the model can be acquire by this method.
 */
const CState* getState() const;

/**
 * Returns the Jacobian Matrix that was calculated.
 * The returned matrix object has the method size
 * to query its size and the methods numCols() and
 * numRows() to query for the number of columns and the
 * number of rows respectively.
 * Individual elements from the matrix can be fetched
 * with the get method which takes the row and the column
 * index as its two arguments.
 * Indices are specified as long values, the return values
 * are of type double.
 */
 const CMatrix<C_FLOAT64>& getJacobian() const;

/**
 * Returns the Jacobian Matrix together with annotations
 * as to which column and row corresponds to what.
 */
const CArrayAnnotation* getJacobianAnnotated() const;

/**
 * Returns the Jacobian matrix for the reduced model
 * (removed mass conservations).
 * The returned matrix object has the method size
 * to query its size and the methods numCols() and numRows()
 * to query for the number of columns and the number of rows
 * respectively.
 * Individual elements from the matrix can be fetched with
```

```
 * the get method which takes the row and the column index
 * as its two arguments.
 * Indices are specified as long values, the return values
 * are of type double.
 */
 const CMatrix<C_FLOAT64>& getJacobianReduced() const;

/**
 * Returns the annotated Jacobian matrix for the reduced model
 * (removed mass conservations).
 */
 const CArrayAnnotation* getJacobianXAnnotated() const;

/**
 * Returns the Eigenvalues for the full model.
 */
const CEigen& getEigenValues() const;

/**
 * Returns the Eigenvalues for the reduced model.
 */
const CEigen& getEigenValuesReduced() const;
```

CSteadyStateMethod.found a steady state was found

CSteadyStateMethod.foundEquilibrium a steady state was found where all reaction fluxes are zero

CSteadyStateMethod.foundNegative a steady state with negative concentrations was found

CSteadyStateMethod::notFound no steady state was found

CSteadyStateMethod::found a steady state was found

CSteadyStateMethod::foundEquilibrium a steady state was found where all reaction fluxes are zero

CSteadyStateMethod::foundNegative a steady state with negative concentrations was found

The class *CEigen* class provides several methods to determine the characteristics of the Eigenvalues.

```
/**
 * Returns the largest real part in all Eigenvalues.
 */
const C_FLOAT& getMaxrealpart() const;

/**
 * Returns the largest imaginary part in all Eigenvalues.
 */
const C_FLOAT64 & getMaximagpart() const;

/**
 * Returns the number of non-zero Eigenvalues.
 */
const C_INT32 & getNzero() const;

/**
 * Returns the stiffness of the Eigenvalues.
 */
const C_FLOAT64 & getStiffness() const;

/**
 * Returns the hierarchy of the Eigenvalues.
```

```
 */
const C_FLOAT64 & getHierarchy() const;

/**
 * Returns the number of Eigenvalues with an
 * imaginary part equal to zero.
 */
const C_INT32 & getNreal() const;

/**
 * Returns the number of Eigenvalues with an
 * imaginary part different from zero.
 */
const C_INT32 & getNimag() const;

/**
 * Get the number of Eigenvalues with
 * a positive real part.
 */
const C_INT32 & getNposreal() const;

/**
 * Get the number of Eigenvalues with
 * a negative real part.
 */
const C_INT32 & getNnegreal() const;

/**
 * Returns a vector with the imaginary parts of
 * all Eigenvalues.
 * The vectors size can be queried with the size method.
 * Individual elements can be acquire with the get method.
 */
const CVector<C_FLOAT64>& getI() const;

/**
 * Returns a vector with the real parts of
 * all Eigenvalues.
 * The vectors size can be queried with the size method.
 * Individual elements can be acquire with the get method.
 */
const CVector<C_FLOAT64>& getR() const;
```

The *CArrayAnnotation* class provides a matrix of values combined with annotation data that provides information on the rows and columns of the matrix. The values of the contained matrix can be acquire the same way as for the float matrix with the *get* method. The method takes two arguments which are the row and the column index. The further methods provided are the following:

```
/**
 * Returns the number of dimensions of the contained matrix.
 */
unsigned int  dimensionality() const;

/**
 * Returns a vector of common names for the given
 * dimension of the matrix.
 */
const std::vector<CRegisteredObjectName>& getAnnotationsCN(unsigned int d) const;

/**
 * Returns a vector of strings that represent the
 * column entries in dimension d. The display
```

```
 * argument determines whether the string contain
 * the display name or the object name of the object
 * represented in a column.
 * A value of true means that display names are returned.
 */
const std::vector<std::string>& getAnnotationsString(unsigned int d, bool display = true)  ↩
    const;

/**
 * Gets a description of what is represented by the
 * dimension given as the argument to the method.
 */
const std::string & getDimensionDescription(unsigned int d) const;

/**
 * Returns a description of the matrix.
 */
const std::string & getDescription() const;
```

The result of the steady state calculation is read directly from the state of the model itself, because after the calculation of the steadystate, the model will contain that state in the transient values of all model entities. So if one wants to find the concentration of some metabolite at the steady state, it can be acquire directly by getting the transient concentration from this metabolite. This works as long as the model has not been changed by adding or removing model entities or by doing any calculations on the model.

1.6.4 Running Parameter Scans

The scan task differs somewhat from the other tasks described so far. For example it is not necessary to set method and problem parameters the way it was done for the time course task.p There are few parameters that have to be set and those can be set by convenience functions. The classes that are needed to run a parameter scan are *CScanTask*, *CScanMethod* and *CScanProblem*. There are three parameters that can be set on the *CScanProblem* instance through methods of that class.

```
/**
 * This method set the subtask for which the scan task
 * can be run. E.g. whether the scan is run on a steady
 * state calculation or on a time course.
 * Allowed types are defined in CCopasiTask.h
 */
void setSubtask(CCopasiTaskType::Type type);

/**
 * Returns the type of the subtask for which the parameter
 * scan is run.
 */
CCopasiTask::Type getSubtask() const;

/**
 * This method sets whether the initial values of the model
 * are set to the result of each run of the subtask or if the
 * initial values should be set to the original initial values
 * of the model.
 * A value of false means always use the original initial
 * values and a value of true means use the result of the last
 * run as initial state for the next run.
 */
 void setAdjustInitialConditions(bool aic);

/**
 * Returns true or false depending on whether the scan will adjust
 * the initial values of the model after each run of the subtask.
```

```
 */
const bool& getAdjustInitialConditions() const;

/**
 * This methods sets whether output from the subtask is
 * collected, or if the output only consists of the end-
 * result of each subtask.
 * Assuming the subtask is a time course simulation,
 * setting this to true will produce a time course for
 * each run of the subtask. If the value is set to false,
 * only the end state of each time course simulation will
 * be written to the report.
 */
void setOutputInSubtask(bool ois);

/**
 * Returns true or false depending on whether output from the
 * subtask is collected or not.
 */
const bool& getOutputInSubtask() const;
```

In order to run a parameter scan, one or more so called scan items have to be defined. Currently COPASI knows three types of scan items: repeat, linear and random. A parameter scan can define several scan items that are stacked and each scan item is applied to the scan item directly beneath it. The simplest scan item is the repeat type. It repeats the scan item (or if it is the lowest scan item it repeats the subtask) below it a certain number of times. No parameter is changed by this scan item type. This can e.g. be used to run a stochastic simulation several times with different random number seeds. (The stochastic time course simulation has a parameter that determines if it is started with a random seed, if this is set to false, the exactly same stochastic simulation will be run several times.) The linear scan item changes an associated model value (usually this will be the initial value of some model entity) within a given range. The stepsize is determined by the number of steps that is given as a parameter. The name of this scan item might be somewhat misleading since it can also change the associated value in a logarithmic fashion rather than in a linear fashion. Last but not least the random scan item can be used to initialize the associated value with a random value. The random value is determined by a permitted range and a distribution function. The scan items are implemented as a nested parameter group. All parameters for the scan items are actually parameters that are stored in a parameter group which represents a scan item. The class *CScanProblem* which itself is also derived from *CCopasiParameterGroup* contains a parameter group called *ScanItems* which holds all the scan items. So in the end the instance of *CScanProblem* holds a parameter group called *ScanItems* which holds parameter groups that represent the individual scan items and those groups hold the parameters that determine the properties of the scan items.

The following list shows the properties of the individual scan item types:

The following example codes shows how to run a simple parameter scan. It sets up the parameter scan to contain one repeat scan item that repeats the specified subtask 10 times:

```
CScanTask* pScanTask=(CScanTask*)pDataModel->addTask(CCopasiTask::scan);
CScanProblem pScanProblem=(CScanProblem*)pScanTask->getProblem();
// set the desired subtask
pScanProblem->setSubtask(CCopasiTask::timeCourse);
// always use the same initial values
pScanProblem->setAdjustInitialConditions(false);
// get the ScanItems parameter group from the problem
CCopasiParameterGroup* pScanItems=pScanProblem->getGroup("ScanItems");
// create a new parameter group to hold a scan item
CCopasiParameterGroup* pScanItem=new CCopasiParameterGroup("scanItem");
// add the parameters "Number of steps", "Type" and "Object" to the
// scan item and set their values
pParameterGroup->addParameter("Number of steps",
                        CCopasiParameter::UINT);
CCopasiParameter* pParameter=pParameterGroup->getParameter("Number of steps");
pParameter->setValue(10);
pParameterGroup->addParameter("Type", CCopasiParameter::UINT);
pParameter=pParameterGroup->getParameter("Type")
```

Parameter name	description	availability
Number of steps (integer)	The number of steps of the scan item. For a repeat or a random scan item this is the number of repetitions, for the linear scan item this is used to determine the stepsize	all scan items
Type (integer)	The type of the scan item (CScanProblem.SCAN_RANDOM, CScanProblem.SCAN_REPEAT or CScanProblem.SCAN_LINEAR)	all scan items
Object (String)	The common name of the object associated with the scan item. For the repeat scan item this is usually an empty string	all scan items
log (boolean)	determines if the value is changed in a logarithmic fashion	linear scan item
Minimum (double)	lower bound	linear and random scan item
Maximum (double)	upper bound	linear and random scan item
Distribution type (integer)	determines which distribution the random numbers follow (CScanProblem.SD_UNIFORM, CScanProblem.SD_GAUSS and CScanProblem.SD_BOLTZ)	random scan item

Table 1.5: scan item parameters

```
pParameter->setValue(CScanProblem::SCAN_REPEAT);
pParameterGroup->addParameter("Object", CCopasiParameter::CN);
pParameter=pParameterGroup->getParameter("Object");
pParameter.setValue("");
try
{
  pScanTask->initialize(CCopasiTask::OUTPUT_COMPLETE, pGobalDataModel, NULL);
  pScanTask->process(true);
  pScanTask->restore();
}
catch(...)
{
    std::cerr << "ERROR: " << Scan failed. << std::endl;;
}
```

It is important to note that the subtask that is going to be used by the parameter scan has to be set up properly prior to running the parameter scan. That is, if one wants to run a parameter scan on a stochastic simulation, the time course task has to be set up so that it runs a stochastic simulation. Also all other parameters like start time and end time etc. have to be specified for the trajectory task before running the parameter scan. The same applies to all other subtasks that are or will be enabled in the future.

For the task describes so far, one was usually interested in the end result of the task. For parameter scans this is usually different. Here one is normally interested in the result of each calculation of the subtask that the parameter scan is being run on. Unfortunately those results are not stored in memory, so right now the only way to get these results is to create a report that stores them to a file. For a documentation on how to create reports and how they are connected to a task see the section on reports Creating Reports.

1.6.5 Running Optimizations

In order to run an optimization, the general procedure is the same as with the other tasks. One has to acquire or create an instance of *COptTask*, choose the desired method to use for the task out of the set of methods COPASI provides for the optimization,

set the desired parameter values on the instance of *COptProblem* that can be fetched from the *COptTask* instance and maybe set some parameter values on the method to change the methods behaviour. At the time of writing, an optimization can either be done with a time course calculation or a steady state calculation. In order to tell the problem which of the two to use, the problem has a method called setSubtaskType. Valid arguments for the method are CCopasiTask.steadyState or CCopasiTask.timeCourse; The parameters that can be set on the methods very much depend on the method that is being used. So if you want to find out which parameters each method provides you will have to read the COPASI manual which if you are lucky does list them. Before running the optimization, two more items have to be set. One is the object that should be optimized, e.g. the initial value of some parameter, the other item is the objective function that the optimization uses the determine the optimal result. The object(s) that are going to be optimized are determined by one or more instances of *COptItem*. Each *COptItem* object stores a reference to the model object to optimize, the start value with which the optimization process will begin and an upper and a lower bound for the value of the object. The optimization method will then try to find values for all objects so that the objective function will (in the optimal case) become minimal. The objective function itself also needs to be specified before the optimization can be run. The objective function is an expression that COPASI tries to minimize by changing the values of the objects referenced in the specified *COptItem*s. In order to set the objective function, the *COptProblem* instance has a method called *setObjectiveFunction*. This method takes a string that represents the mathematical expression of the function. The mathematical expression can contain all elements that can be used to describe expressions as described in the COPASI manual. Model objects that are used in the expression have to be referenced with their so called common name enclosed in angular brackets (<CN>). To get the common name from an object derived from *CCopasiObject* the method *getCN* can be used.

Here a small example that runs a simple optimization task:

```
// create the optimization task
COptTask* pOptTask=(COptTask*)pDataModel->addTask(CCopasiTask::optimization);
if(pOptTask==NULL) return NULL;

// use the Levenberg-Marquardt method for the optimization
pOptTask->setMethodType(CCopasiMethod::LevenbergMarquardt);

// acquire the problem instance from the task
COptProblem* pOptProblem=(COptProblem*)pOptTask->getProblem();
if(pOptProblem==NULL) return NULL;

// create the objective function
// in this case the objective function is the transient
// value of some model parameter
// so COPASI will try to find values for all COptItems
// so that the transient value of the parameter after
// the simulation is minimal
CCopasiObjectName ref=CCopasiObjectName("Reference=Value");
CCopasiObject* pObject=pModelValue->getObject(ref);
std:string objectiveFunction=pObject->getCN();
objectiveFunction="<"+objectiveFunction+">";
// set the objective function on the problem
pOptProblem->setObjectiveFunction(objectiveFunction);
// set the problems subtask type to indicate that the optimization
// is to be run on a time course simulation
pOptProblem->setSubtaskType(CCopasiTask::timeCourse);

// create an optimization item for the
// initial value of some model parameter
ref=CCopasiObjectName("Reference=InitialValue");
pObject=pSomeFixedModelValue->getObject(ref);
CCopasiObjectName optObjectName=pObject->getCN();
COptItem* pOptItem2=pOptProblem->addOptItem(optObjectName);
pOptItem2->setStartValue(1.0,datamodel);
pOptItem2->setLowerBound(0.0,datamodel);
pOptItem2->setUpperBound(2.0,datamodel);

// change some of the parameters of the method
COptMethod* pOptMethod=(COptMethod*)pOptTask->getMethod();
```

```
CCopsiParameter* pParam=pOptMethod->getParameter("Iteration Limit");
pParam->setValue(2000);
pParam=pOptMethod->getParameter("Tolerance");
pParam->setValue(1.0e-5);
try
{
    // run the optimization
  pOptTask->initialize(CCopasiTask::OUTPUT_COMPLETE, pGobalDataModel, NULL);
  pOptTask->process(true);
  pOptTask->restore();
}
catch(...)
{
  std::cerr << "ERROR: Optimization failed." << std::endl;
}
```

After the optimization has been run, the solution of the optimization is stored in the problem instance. In order to get the lowest value COPASI was able to find for the objective function, use the call to *getSolutionValue*. The values for the *COptItem* objects corresponding to that result, use *getSolutionVariables*. This method returns a container with double values. The size of the container can be queried with its *size* method and individual values can be acquire by calling *get* with the index of the value. The value for index *i* belongs to the *COptItem* with index *i* in the list of *COptItem*s. To get a container with all defined *COptItem*s, use *getOptItemList*.

1.6.6 Fitting parameters

Parameter fitting can be used to determine the value of certain model entities with the help of experimental data. The API for doing parameter fitting is very similar to the one for running an optimization (see above). This is because the classes for parameter fitting are derived from the optimization classes. This makes sense since the actions for running an optimization are very similar to what COPASI does when doing parameter fitting. The major difference is that for an optimization, the user has to provide an objective function that is minimized whereas for the parameter fitting, the user provides experimental data and some information about the data and COPASI determines the objective function from that information.

The first step when doing parameter fitting is to get an instance of *CFitTask*. Either there is already one in the list of tasks which can be used, or we simplify add one, as demonstrated in the example below. The *CFitTask* has the methods *getMethod* and *getProblem* which provide the method and the problem for the task.

The methods that can be used for parameter fitting should be the same set of methods that is available for the optimization. The methods that are available can be queried with *getValidMethods* in *CFitTask* and a specific method can be set with *setMethodType* also in *CFitTask*.

Most settings for the parameter fitting however are done in the problem. The problem contains a parameter group called *Experiment Set* which contains the information about the experimental data against which the parameters should be fitted. The information for each experiment is stored in an instance of *CExperiment*. The information that needs to be provided for each experment is

1. the name of the file where the experimental data can be found
2. the row number where the data for the experiment starts
3. the row number where the data for the experiment ends
4. the type of experiment that the data represents (time course or steady state)
5. the number of data columns in the data set
6. the mapping of the columns to the entities in the model and their role

Another useful piece of information that can be set for the experiment is the separator that is used in the data file to separate the individual columns. This separator can be set with the *setSeparator* method in *CExperiment*. It takes a string as its only argument. One thing to watch out for is the fact that COPASI tries to match the separator exactly. That is, if the separator is set

to be one space character, having two space characters between two columns in the file will lead COPASI to think that there is an empty column. Unfortunately there is no way right now to tell COPASI that the delimiter for the columns is e.g. one or more space characters.

The information about the mapping from data columns to model entities is stored in an instance of *CExperimentObjectmap* which can be aquired with the *getObjectMap* method from *CExperiment*. After the information for the experiment has been set, the experiment needs to be added to the experiment set. The *addExperiment* method creates a copy of the experiment that is added, so make sure that you get the experiment from the experiment set before you make further modifications or use it, e.g. to add it to the fit item (see below).

Next we need to create the fit items which correspond to the optimization items in the optimization task. Fit items are instances of *CFitItem* and they have an upper bound, a lower bound and a start value just like optimization items. By default each fit item is influenced by all experiments. if you want to limit a fit item to be only influenced by certain experiments, you have to add those experiments to the fit item with the *addExperiment* method of *CFitItem*.

There is no specialized method to add fit items to the fit problem, so we have to use the generic methods provided be *CCopasiParameterGroup* to do this. The fit problem contains a parameter group called *OptimizationItemList* which holds all the fit items. Since the problem itself is derived from *CCopasiParameterGroup*, we can use the *getParameter* method with the name of the parameter in order to get the parameter group. Now we can add the fit items with the *addParameter* method to the group. Just like the *addExperiment* method above, this method makes a copy of its argument, so if you are planning to use the fit item again, be sure to get the fit item from the group again.

A fit problem can actually contain many experiments and many fit items, but the example code below only uses one experiment and one fit item to fit the reaction parameter of a simple system. The code assumes that the models consists of only one compartment with two species named *A* and *B* as well as one reaction with a local reaction parameter (which is fitted).

```
CFitTask* pFitTask= (CFitTask*)pDataModel->addTask(CCopasiTask::parameterFitting);
// the method in a fit task is an instance of COptMethod or a subclass of
// it.
COptMethod pFitMethod=(COptMethod*)pFitTask->getMethod();
// the object must be an instance of COptMethod or a subclass thereof
// (CFitMethod)
CFitProblem* pFitProblem=(CFitProblem*)pFitTask->getProblem();
CExperimentSet* pExperimentSet=&(CExperimentSet*)pFitProblem->getParameter("Experiment Set ↩
    ");
// first experiment
CExperiment* pExperiment=new CExperiment(pDataModel);
pExperiment->setFileName("parameter_fitting_data_simple.txt");
pExperiment->setFirstRow(1);
pExperiment->setLastRow(22);
pExperiment->setHeaderRow(1);
pExperiment->setExperimentType(CCopasiTask::timeCourse);
pExperiment->setNumColumns(3);
CExperimentObjectMap* pObjectMap=&pExperiment->getObjectMap();
bool result=pObjectMap->setNumCols(3);
result=pObjectMap->setRole(0,CExperiment::time);
CModel* pModel=pDataModel->getModel();
// create the common name for time (column 0)
CCopasiObject* pTimeReference=pModel->getObject(CCopasiObjectName("Reference=Time"));
pObjectMap->setObjectCN(0,pTimeReference->getCN());
CMetab* pMetabA=(*pModel->getMetabolites())[0];
CMetab* pMetabB=NULL;
if(pMetabA->getObjectName()!="A")
{
    pMetabB=pMetabA;
    pMetabA=(*pModel->getMetabolites())[1];
}
else
{
    pMetabB=(*pModel->getMetabolites())[1];
}
// set the role to dependent
pObjectMap->setRole(1,CExperiment::dependent);
```

```
// create the common name for the particle number of A (column 1)
CCopasiObject* pParticleReference=pMetabA->getObject(CCopasiObjectName("Reference= ↩
    ParticleNumber"));
pObjectMap->setObjectCN(1,pParticleReference->getCN());
// set the role to dependent
pObjectMap->setRole(2,CExperiment::dependent);
// create the common name for the particle number of B (column 1)
pParticleReference=pMetabB->getObject(CCopasiObjectName("Reference=ParticleNumber"));
pObjectMap->setObjectCN(2,pParticleReference->getCN());
// getObjectCN returns a string whereas getCN returns a
// CCopasiObjectName
pExperimentSet->addExperiment(pExperiment);
// addExperiment makes a copy, so we need to get the added experiment
// again
delete pExperiment;
pExperiment=pExperimentSet->getExperiment(0);

CReaction* pReaction=(*pModel->getReactions())[0];
CCopasiParameter* pParameter=pReaction->getParameters()->getParameter(0);

// define CFitItems
CCopasiObject* pParameterReference=pParameter->getObject(CCopasiObjectName("Reference=Value ↩
    "));
CFitItem* pFitItem=new CFitItem(pDataModel);
pFitItem->setObjectCN(pParameterReference->getCN());
pFitItem->setStartValue(4.0);
pFitItem->setLowerBound(CCopasiObjectName("0.0001"),datamodel);
pFitItem->setUpperBound(CCopasiObjectName("10"),datamodel);
// add the fit item to the correct parameter group
CCopasiParameterGroup* pOptimizationItemGroup=(CCopasiParameterGroup*)pFitProblem-> ↩
    getParameter("OptimizationItemList");
pOptimizationItemGroup->addParameter(pFitItem);
// addParameter makes a copy of the fit item, so we have to get it back
delete pFitItem;
pFitItem=(CFitItem*)pOptimizationItemGroup->getParameter(0);
result=true;
try
{
  pFitTask->initialize(CCopasiTask::OUTPUT_COMPLETE, pGobalDataModel, NULL);
  result=pFitTask->process(true);
  pFitTask->restore();
}
catch(...)
{
  result=false;
}
// get the fitted value for the reaction parameter
double fittedK=pFitItem->getLocalValue();
```

After running the fitting task, the fit result is stored in the individual fit items. The best value that was achived during parameter fitting can be queried by the *getLocalValue* method from each fit item.

1.7 Creating Reports

Sometimes it is necessary to write the results of a calculation to a file instead of keeping it in memory. In order to achieve this, a report for the results has to be created and this report has to be associated with a task. In COPASI reports are represented by the class *CReport*. A report stored a so called report definition that specifies what is being written and it contains a filename that determines where the report is written to. In order to set the filename, you can use the method *setTarget* with the path of the file as the argument. It is up to the programmer to make sure that the directory part of the path exists and that it is writable. To set the

report definition of a report instance, use the method *setReportDefinition*. The only argument to the call is an instance of class *CReportDefinition*. A report definition can work in two modes. In the default mode, the report definition is a table and all objects that are added to the report definition are written separated by a user definable separator. In the second mode a report definitions consists of three parts a header, a body and a footer. The part that is specified as the header is written before the associated task is executed, the body is executed for each step of the associated task and the footer is finally written after the task has finished. Each part is a container that holds zero or more report definition items. To get the individual containers, the *CReportDefinition* class has the methods *getTableAddr* if the report definition is in table mode or *getHeaderAddr*, *getBodyAddr* and *getFooterAddr* if it is not in table mode. To specify whether a report definition is to run in table mode, you can use the method *setIsTable* which takes a boolean value that determines whether the report definition is a table or not. The flag can be queried with the *isTable* method. The individual items are of type *CRegisteredObjectName*. A new instance of *CRegisteredObjectName* can be created by calling the constructor with a string that represents the common name of the object that is to written. If the registered object name should represent a model entity, the string has to be the common name of the object. If you want to add an element the report that represents a simple stirng, you have to create an instance of *CCopasiStaticString* by calling the constructor of this class with the string that you want to add to the report definition. Once you have an instance of *CCopasiStaticString*, you can create an instance of *CRegisteredObjectName* the same way as it is done for other objects, you take the string representation of the common name of your instance of *CCopasiStaticString* and use it as the argument to the constructor of *CRegisteredObjectName*.

In COPASI the reports definitions are stored in the instances of *CCopasiDataModel*. To get this list use the method *getReportDefinitionList* on the instance of *CCopasiDataModel*.

```
pDataModel->getReportDefinitionList();
```

The returned object is a container that stores all report definitions. To create a new report definition, use *createReportDefinitions* which is a method of the container. The two arguments to the call are the name of the report definition and a description. Once you have a report definition object, you can use *setTaskType* to set the type of task the report is intended for. The method takes one argument which is one of the task types defined in *CCopasiTask*, e.g. *CCopasiTask.timeCourse* for the time course task. Next you can fill the tree sections (header, body, footer) of the report definition or the table with whatever has to be written to the output. Once the report definition is finished, we have to tell the task to use it. This is done through the tasks report object which can be acquire with the *getReport()* method of *CCopasiTask*. The returned report object has a method *setReportDefinition* which takes the report definition object as its argument. Last but not least, we have to specify a filename where the report is written to which is done with the *setTarget* method of the same report object. The following example shows how to create a report for a time course simulation that writes the values of all global parameters as an xhtml file with a table for the values:

```
...
CReportDefinitionVector* pReportDefs=pDataModel->getReportDefinitionList();
CReportDefinition* pRepDef=pReportDefs->createReportDefinition("htmlConc",
                                    "value table in HTML format");
pRepDef->setIsTable(false);
CCopasiStaticString htmlHeader=CCopasiStaticString("<?xml version=\"1.0\""
 + " encoding=\"UTF-8\"?>\n<!DOCTYPE html PUBLIC "
 + "\"-//W3C//DTD XHTML 1.0 Transitional//EN\" "
 + "\"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd\">\n"
 + "<html xmlns=\"http://www.w3.org/1999/xhtml\" xml:lang=\"en\""
 + "lang=\"en\">\n<body>\n<table>\n");
CCopasiStaticString htmlFooter=CCopasiStaticString(
                                     "</table>\n</body>\n</html>\n");
std::vector<CRegisteredObjectName>* pHeader=pRepDef->getHeaderAddr();
pHeader->add(CRegisteredObjectName(htmlHeader.getCN()));
std::vector<CRegisteredObjectName>* pFooter=pRepDef->getFooterAddr();
pFooter->add(new CRegisteredObjectName(htmlFooter.getCN()));
std::vector<CRegisteredObjectName>* pBody=pRepDef->getBodyAddr();
CModel* pModel=pDataModel->getModel();
CCopasiObjectName name=CCopasiObjectName("Reference=Time");
CCopasiObject* pTimeObject=pModel->getObject(name);
CCopasiStaticString s=CCopasiStaticString("<tr>\n<td>");
pBody->add(CRegisteredObjectName(s.getCN()));
pBody->add(CRegisteredObjectName(pTimeObject->getCN()));
s=CCopasiStaticString("</td>\n");
pBody->add(CRegisteredObjectName(s.getCN()));
name=CCopasiObjectName("Reference=Value")
```

```
long i,iMax=pModel->getModelValues()->size();
for(i=0;i<iMax;++i)
{
    s=CCopasiStaticString("<td>");
    pBody->add(CRegisteredObjectName(s.getCN()));
    CModelValue* pMV=(*pModel->getModelValues())[i];
    CCopasiObject* pValueObject=pMV->getObject(name);
    pBody->add(CRegisteredObjectName(pValueObject->getCN()));
    s=CCopasiStaticString("</td>\n");
    pBody->add(CRegisteredObjectName(s.getCN()));
}
s=CCopasiStaticString("</tr>\n");
pBody->add(CRegisteredObjectName(s.getCN()));
pRepDef->setTaskType(CCopasiTask::timeCourse);
CReport* pReport=&pTask->getReport();
pReport->setReportDefinition(pRepDef);
pReport->setAppend(false);
pReport->setTarget("table.xhtml");
 ...
```

1.8 Error Messages

COPASI uses a message deque to store certain warning and error messages that occur during a process, e.g. during import and export of SBML files. The methods to work with the message deque are part of the CCopasiMessage class.

It is always a good idea to check the message deque after doing complex things like running a certain task, or loading models.

The following methods are available to work with the message deque in COPASI.

```
/**
 * This static method peeks at the first message created in COPASI.
 * If no more messages are in the deque the message
 * (MCCopasiMessage + 1, "Message (1): No more messages." is returned.
 */
static const CCopasiMessage& peekFirstMessage();

/**
 * This static method peeks at the last message created in COPASI.
 * If no more messages are in the deque the message
 * (MCCopasiMessage + 1, "Message (1): No more messages." is returned.
 */
static const CCopasiMessage& peekLastMessage();

/**
 * This static method retrieves the first message created in COPASI.
 * Consecutive calls allow for the retrieval of all generated
 * messages in chronological order. If no more messages are in
 * the deque the message (MCCopasiMessage + 1, "Message (1):
 * No more messages." is returned.
 */
static CCopasiMessage getFirstMessage();

/**
 * This static method retrieves the last message created in COPASI.
 * Consecutive calls allow for the retrieval of all generated
 * messages in reverse chronological order. If no more messages
 * are in the deque the message (MCCopasiMessage + 1, "Message
 * (1): No more messages." is returned.
 */
static CCopasiMessage getLastMessage();
```

```
/**
 * Static method that retrieves the text of all messages in the deque in chronological
 * or reverse chronological order. If more than on message is in
 * the deque the messages are seperated by an empty line.
 */
static std::string getAllMessageText(const bool& chronological = true);

/**
 * Static method that clears the message deque.
 */
static void clearDeque();

/**
 * Static method that retreives the size of the message deque
 */
static unsigned C_INT32 size();

/**
 * Static methosd that retrieves the highest severity of the messages in the deque.
 * The returned severity can be CCopasiMessage::RAW, CCopasiMessage::TRACE,
 * CCopasiMessage::COMMANDLINE, CCopasiMessage::WARNING, CCopasiMessage::ERROR,
 * CCopasiMessage::EXCEPTION, CCopasiMessage::RAW_FILTERED,
 * CCopasiMessage::TRACE_FILTERED, CCopasiMessage::COMMANDLINE_FILTERED,
 * CCopasiMessage::WARNING_FILTERED, CCopasiMessage::ERROR_FILTERED or
 * CCopasiMEssage::EXCEPTION_FILTERED.
 *
 * The most common of those are WARNING, ERROR or EXCEPTION.
 */
static CCopasiMessage::Type getHighestSeverity();

/**
 * Static method that checks whether a message with the given number is in the deque
 */
static bool checkForMessage(const unsigned C_INT32& number);

/**
 *  Retrieves the text of the message instance.
 */
const std::string& getText() const;

/**
 *  Retrieves the type of the message instance.
 */
const CCopasiMessage::Type& getType() const;

/**
 *  Retrieves the number of the message instance.
 */
const unsigned C_INT32& getNumber() const;
```

1.9 Programming Examples

This section contains some programming examples that might be helpful. We hope that eventually these examples will cover most of the backend.

1.9.1 Creating and saving a model

This examples shows how to build a simple model with the copasi backend and how it can be saved as either a COPASI file or as an SBML file.

```
// Begin CVS Header
//   $Source: /fs/turing/cvs/copasi_dev/copasi/bindings/cpp_examples/example1/example1.cpp, ↩
    v $
//   $Revision: 1.1.6.1 $
//   $Name:  $
//   $Author: gauges $
//   $Date: 2011/09/15 14:19:53 $
// End CVS Header

// Copyright (C) 2011 by Pedro Mendes, Virginia Tech Intellectual
// Properties, Inc., University of Heidelberg, and The University
// of Manchester.
// All rights reserved.

// Copyright (C) 2008 by Pedro Mendes, Virginia Tech Intellectual
// Properties, Inc., EML Research, gGmbH, University of Heidelberg,
// and The University of Manchester.
// All rights reserved.

/**
 * This is an example on how to build models with the COPASI backend API.
 */
#include <iostream>
#include <vector>
#include <string>
#include <set>


#define COPASI_MAIN

#include "copasi/copasi.h"
#include "copasi/report/CCopasiRootContainer.h"
#include "copasi/CopasiDataModel/CCopasiDataModel.h"
#include "copasi/model/CModel.h"
#include "copasi/model/CCompartment.h"
#include "copasi/model/CMetab.h"
#include "copasi/model/CReaction.h"
#include "copasi/model/CChemEq.h"
#include "copasi/model/CModelValue.h"
#include "copasi/function/CFunctionDB.h"
#include "copasi/function/CFunction.h"
#include "copasi/function/CEvaluationTree.h"


int main()
{
  // initialize the backend library
  // since we are not interested in the arguments
  // that are passed to main, we pass 0 and NULL to
  // init
  CCopasiRootContainer::init(0, NULL);
  assert(CCopasiRootContainer::getRoot() != NULL);
  // create a new datamodel
  CCopasiDataModel* pDataModel = CCopasiRootContainer::addDatamodel();
  assert(CCopasiRootContainer::getDatamodelList()->size() == 1);
  // get the model from the datamodel
  CModel* pModel = pDataModel->getModel();
```

```
assert(pModel != NULL);
// set the units for the model
// we want seconds as the time unit
// microliter as the volume units
// and nanomole as the substance units
pModel->setTimeUnit(CModel::s);
pModel->setVolumeUnit(CModel::microl);
pModel->setQuantityUnit(CModel::nMol);

// we have to keep a set of all the initial values that are changed during
// the model building process
// They are needed after the model has been built to make sure all initial
// values are set to the correct initial value
std::set<const CCopasiObject*> changedObjects;

// create a compartment with the name cell and an initial volume of 5.0
// microliter
CCompartment* pCompartment = pModel->createCompartment("cell", 5.0);
const CCopasiObject* pObject = pCompartment->getObject(CCopasiObjectName("Reference= ↩
    InitialVolume"));
assert(pObject != NULL);
changedObjects.insert(pObject);
assert(pCompartment != NULL);
assert(pModel->getCompartments().size() == 1);
// create a new metabolite with the name glucose and an inital
// concentration of 10 nanomol
// the metabolite belongs to the compartment we created and is is to be
// fixed
CMetab* pGlucose = pModel->createMetabolite("glucose", pCompartment->getObjectName(),  ↩
    10.0, CMetab::FIXED);
pObject = pGlucose->getObject(CCopasiObjectName("Reference=InitialConcentration"));
assert(pObject != NULL);
changedObjects.insert(pObject);
assert(pCompartment != NULL);
assert(pGlucose != NULL);
assert(pModel->getMetabolites().size() == 1);
// create a second metabolite called glucose-6-phosphate with an initial
// concentration of 0. This metabolite is to be changed by reactions
CMetab* pG6P = pModel->createMetabolite("glucose-6-phosphate", pCompartment-> ↩
    getObjectName(), 0.0, CMetab::REACTIONS);
assert(pG6P != NULL);
pObject = pG6P->getObject(CCopasiObjectName("Reference=InitialConcentration"));
assert(pObject != NULL);
changedObjects.insert(pObject);
assert(pModel->getMetabolites().size() == 2);
// another metabolite for ATP, also fixed
CMetab* pATP = pModel->createMetabolite("ATP", pCompartment->getObjectName(), 10.0,  ↩
    CMetab::FIXED);
assert(pATP != NULL);
pObject = pATP->getObject(CCopasiObjectName("Reference=InitialConcentration"));
assert(pObject != NULL);
changedObjects.insert(pObject);
assert(pModel->getMetabolites().size() == 3);
// and one for ADP
CMetab* pADP = pModel->createMetabolite("ADP", pCompartment->getObjectName(), 0.0, CMetab ↩
    ::REACTIONS);
assert(pADP != NULL);
pObject = pADP->getObject(CCopasiObjectName("Reference=InitialConcentration"));
assert(pObject != NULL);
changedObjects.insert(pObject);
assert(pModel->getMetabolites().size() == 4);
// now we create a reaction
```

```
CReaction* pReaction = pModel->createReaction("hexokinase");
assert(pReaction != NULL);
assert(pModel->getReactions().size() == 1);
// hexokinase converts glucose and ATP to glucose-6-phosphate and ADP
// we can set these on the chemical equation of the reaction
CChemEq* pChemEq = &pReaction->getChemEq();
// glucose is a substrate with stoichiometry 1
pChemEq->addMetabolite(pGlucose->getKey(), 1.0, CChemEq::SUBSTRATE);
// ATP is a substrate with stoichiometry 1
pChemEq->addMetabolite(pATP->getKey(), 1.0, CChemEq::SUBSTRATE);
// glucose-6-phosphate is a product with stoichiometry 1
pChemEq->addMetabolite(pG6P->getKey(), 1.0, CChemEq::PRODUCT);
// ADP is a product with stoichiometry 1
pChemEq->addMetabolite(pADP->getKey(), 1.0, CChemEq::PRODUCT);
assert(pChemEq->getSubstrates().size() == 2);
assert(pChemEq->getProducts().size() == 2);
// this reaction is to be irreversible
pReaction->setReversible(false);
assert(pReaction->isReversible() == false);
// now we ned to set a kinetic law on the reaction
// maybe constant flux would be OK
// we need to get the function from the function database
CFunctionDB* pFunDB = CCopasiRootContainer::getFunctionList();
assert(pFunDB != NULL);
// it should be in the list of suitable functions
// lets get all suitable functions for an irreversible reaction with  2 substrates
// and 2 products
std::vector<CFunction*> suitableFunctions = pFunDB->suitableFunctions(2, 2, TriFalse);
assert(!suitableFunctions.empty());
std::vector<CFunction*>::iterator it = suitableFunctions.begin(), endit =  ↩
    suitableFunctions.end();

while (it != endit)
  {
    // we just assume that the only suitable function with Constant in
    // it's name is the one we want
    if ((*it)->getObjectName().find("Constant") != std::string::npos)
      {
        break;
      }

    ++it;
  }

if (it != endit)
  {
    // we set the function
    // the method should be smart enough to associate the reaction entities
    // with the correct function parameters
    pReaction->setFunction(*it);
    assert(pReaction->getFunction() != NULL);
    // constant flux has only one function parameter
    assert(pReaction->getFunctionParameters().size() == 1);
    // so there should be only one entry in the parameter mapping as well
    assert(pReaction->getParameterMappings().size() == 1);
    CCopasiParameterGroup* pParameterGroup = &pReaction->getParameters();
    assert(pParameterGroup->size() == 1);
    CCopasiParameter* pParameter = pParameterGroup->getParameter(0);
    // make sure the parameter is a local parameter
    assert(pReaction->isLocalParameter(pParameter->getObjectName()));
    // now we set the value of the parameter to 0.5
    pParameter->setValue(0.5);
```

```
    pObject = pParameter->getObject(CCopasiObjectName("Reference=Value"));
    assert(pObject != NULL);
    changedObjects.insert(pObject);
  }
else
  {
    std::cerr << "Error. Could not find a kinetic law that conatins the term \"Constant ↩
        \"." << std::endl;
    return 1;
  }

// now we also create a separate reaction for the backwards reaction and
// set the kinetic law to irreversible mass action
// now we create a reaction
pReaction = pModel->createReaction("hexokinase-backwards");
assert(pReaction != NULL);
assert(pModel->getReactions().size() == 2);
pChemEq = &pReaction->getChemEq();
// glucose is a product with stoichiometry 1
pChemEq->addMetabolite(pGlucose->getKey(), 1.0, CChemEq::PRODUCT);
// ATP is a product with stoichiometry 1
pChemEq->addMetabolite(pATP->getKey(), 1.0, CChemEq::PRODUCT);
// glucose-6-phosphate is a substrate with stoichiometry 1
pChemEq->addMetabolite(pG6P->getKey(), 1.0, CChemEq::SUBSTRATE);
// ADP is a substrate with stoichiometry 1
pChemEq->addMetabolite(pADP->getKey(), 1.0, CChemEq::SUBSTRATE);
assert(pChemEq->getSubstrates().size() == 2);
assert(pChemEq->getProducts().size() == 2);
// this reaction is to be irreversible
pReaction->setReversible(false);
assert(pReaction->isReversible() == false);
// now we ned to set a kinetic law on the reaction
CFunction* pMassAction = dynamic_cast<CFunction*>(pFunDB->findFunction("Mass action ( ↩
    irreversible)"));
assert(pMassAction != NULL);
// we set the function
// the method should be smart enough to associate the reaction entities
// with the correct function parameters
pReaction->setFunction(pMassAction);
assert(pReaction->getFunction() != NULL);

assert(pReaction->getFunctionParameters().size() == 2);
// so there should be two entries in the parameter mapping as well
assert(pReaction->getParameterMappings().size() == 2);
// mass action is a special case since the parameter mappings for the
// substrates (and products) are in a vector

// Let us create a global parameter that is determined by an assignment
// and that is used as the rate constant of the mass action kinetics
// it gets the name rateConstant and an initial value of 1.56
CModelValue* pModelValue = pModel->createModelValue("rateConstant", 1.56);
assert(pModelValue != NULL);
pObject = pModelValue->getObject(CCopasiObjectName("Reference=InitialValue"));
assert(pObject != NULL);
changedObjects.insert(pObject);
assert(pModel->getModelValues().size() == 1);
// set the status to assignment
pModelValue->setStatus(CModelValue::ASSIGNMENT);
// the assignment does not have to make sense
pModelValue->setExpression("1.0 / 4.0 + 2.0");

// now we have to adjust the parameter mapping in the reaction so
```

```
  // that the kinetic law uses the global parameter we just created instead
  // of the local one that is created by default
  // The first parameter is the one for the rate constant, so we point it to
  // the key of out model value
  pReaction->setParameterMapping(0, pModelValue->getKey());
  // now we have to set the parameter mapping for the substrates
  pReaction->addParameterMapping("substrate", pG6P->getKey());
  pReaction->addParameterMapping("substrate", pADP->getKey());

  // finally compile the model
  // compile needs to be done before updating all initial values for
  // the model with the refresh sequence
  pModel->compileIfNecessary(NULL);

  // now that we are done building the model, we have to make sure all
  // initial values are updated according to their dependencies
  std::vector<Refresh*> refreshes = pModel->buildInitialRefreshSequence(changedObjects);
  std::vector<Refresh*>::iterator it2 = refreshes.begin(), endit2 = refreshes.end();

  while (it2 != endit2)
    {
      // call each refresh
      (**it2)();
      ++it2;
    }

  // save the model to a COPASI file
  // we save to a file named example1.cps, we don't want a progress report
  // and we want to overwrite any existing file with the same name
  // Default tasks are automatically generated and will always appear in cps
  // file unless they are explicitley deleted before saving.
  pDataModel->saveModel("example1.cps", NULL, true);

  // export the model to an SBML file
  // we save to a file named example1.xml, we want to overwrite any
  // existing file with the same name and we want SBML L2V3
  pDataModel->exportSBML("example1.xml", true, 2, 3);
  pDataModel->exportSBML("example1.xml", true, 3, 1);

  // destroy the root container once we are done
  CCopasiRootContainer::destroy();
}
```

1.9.2 Loading and processing a model

This example shows how a COPASI file can be loaded and the individual model elements can be accessed.

```
// Begin CVS Header
//   $Source: /fs/turing/cvs/copasi_dev/copasi/bindings/cpp_examples/example2/example2.cpp, ↩
    v $
//   $Revision: 1.1.6.1 $
//   $Name: Build-33 $
//   $Author: shoops $
//   $Date: 2011/01/13 19:36:31 $
// End CVS Header

// Copyright (C) 2010 by Pedro Mendes, Virginia Tech Intellectual
// Properties, Inc., University of Heidelberg, and The University
// of Manchester.
```

```
// All rights reserved.

// Copyright (C) 2008 by Pedro Mendes, Virginia Tech Intellectual
// Properties, Inc., EML Research, gGmbH, University of Heidelberg,
// and The University of Manchester.
// All rights reserved.

/**
 * This is an example on how to load a cps file
 * and output some information on the model
 */

#include <iostream>
#include <string>

#define COPASI_MAIN

#include "copasi/copasi.h"
#include "copasi/report/CCopasiRootContainer.h"
#include "copasi/CopasiDataModel/CCopasiDataModel.h"
#include "copasi/model/CModel.h"
#include "copasi/model/CCompartment.h"
#include "copasi/model/CMetab.h"
#include "copasi/model/CReaction.h"

int main(int argc, char** argv)
{
  // initialize the backend library
  CCopasiRootContainer::init(argc, argv);
  assert(CCopasiRootContainer::getRoot() != NULL);
  // create a new datamodel
  CCopasiDataModel* pDataModel = CCopasiRootContainer::addDatamodel();
  assert(CCopasiRootContainer::getDatamodelList()->size() == 1);

  // the only argument to the main routine should be the name of a CPS file
  if (argc == 2)
    {
      std::string filename = argv[1];

      try
        {
          // load the model without progress report
          pDataModel->loadModel(filename, NULL);
        }
      catch (...)
        {
          std::cerr << "Error while loading the model from file named \"" << filename <<  ↩
              "\"." << std::endl;
          CCopasiRootContainer::destroy();
          return 1;
        }

      CModel* pModel = pDataModel->getModel();
      assert(pModel != NULL);
      std::cout << "Model statistics for model \"" << pModel->getObjectName() << "\"." <<  ↩
          std::endl;

      // output number and names of all compartments
      size_t i, iMax = pModel->getCompartments().size();
      std::cout << "Number of Compartments: " << iMax << std::endl;
      std::cout << "Compartments: " << std::endl;
```

```
      for (i = 0; i < iMax; ++i)
        {
          CCompartment* pCompartment = pModel->getCompartments()[i];
          assert(pCompartment != NULL);
          std::cout << "\t" << pCompartment->getObjectName() << std::endl;
        }

      // output number and names of all metabolites
      iMax = pModel->getMetabolites().size();
      std::cout << "Number of Metabolites: " << iMax << std::endl;
      std::cout << "Metabolites: " << std::endl;

      for (i = 0; i < iMax; ++i)
        {
          CMetab* pMetab = pModel->getMetabolites()[i];
          assert(pMetab != NULL);
          std::cout << "\t" << pMetab->getObjectName() << std::endl;
        }

      // output number and names of all reactions
      iMax = pModel->getReactions().size();
      std::cout << "Number of Reactions: " << iMax << std::endl;
      std::cout << "Reactions: " << std::endl;

      for (i = 0; i < iMax; ++i)
        {
          CReaction* pReaction = pModel->getReactions()[i];
          assert(pReaction != NULL);
          std::cout << "\t" << pReaction->getObjectName() << std::endl;
        }
    }
  else
    {
      std::cerr << "Usage: example2 CPSFILE" << std::endl;
      CCopasiRootContainer::destroy();
      return 1;
    }

  // clean up the library
  CCopasiRootContainer::destroy();
}
```

1.9.3 Running a timecourse simulation

This example shows how a SBML file can be imported and a time course simulation can be run on the model. It also demonstrates how the results from the time course simulation can be accessed.

The example also shows how a report can be defined from the backend in order to get output to a file.

```
// Begin CVS Header
//   $Source: /fs/turing/cvs/copasi_dev/copasi/bindings/cpp_examples/example3/example3.cpp, ↩
    v $
//   $Revision: 1.1.6.4 $
//   $Name: Build-33 $
//   $Author: gauges $
//   $Date: 2011/12/16 07:06:51 $
// End CVS Header

// Copyright (C) 2011 - 2010 by Pedro Mendes, Virginia Tech Intellectual
// Properties, Inc., University of Heidelberg, and The University
```

```
// of Manchester.
// All rights reserved.

// Copyright (C) 2008 by Pedro Mendes, Virginia Tech Intellectual
// Properties, Inc., EML Research, gGmbH, University of Heidelberg,
// and The University of Manchester.
// All rights reserved.

/**
 * This is an example on how to import an sbml file
 * create a report for a time course simulation
 * and run a time course simulation
 */

#include <iostream>
#include <string>

#define COPASI_MAIN

#include "copasi/copasi.h"
#include "copasi/report/CCopasiRootContainer.h"
#include "copasi/CopasiDataModel/CCopasiDataModel.h"
#include "copasi/model/CModel.h"
#include "copasi/model/CMetab.h"
#include "copasi/report/CReport.h"
#include "copasi/report/CReportDefinition.h"
#include "copasi/report/CReportDefinitionVector.h"
#include "copasi/trajectory/CTrajectoryTask.h"
#include "copasi/trajectory/CTrajectoryMethod.h"
#include "copasi/trajectory/CTrajectoryProblem.h"
#include "copasi/trajectory/CTimeSeries.h"
#include "copasi/function/CFunctionDB.h"


int main(int argc, char** argv)
{
  // initialize the backend library
  CCopasiRootContainer::init(argc, argv);
  assert(CCopasiRootContainer::getRoot() != NULL);
  // create a new datamodel
  CCopasiDataModel* pDataModel = CCopasiRootContainer::addDatamodel();
  assert(CCopasiRootContainer::getDatamodelList()->size() == 1);

  // the only argument to the main routine should be the name of an SBML file
  if (argc == 2)
    {
      std::string filename = argv[1];

      try
        {
          // load the model without progress report
          pDataModel->importSBML(filename, NULL);
          //pDataModel->loadModel(filename, NULL);
        }
      catch (...)
        {
          std::cerr << "Error while importing the model from file named \"" << filename <<  ↩
              "\"." << std::endl;
          CCopasiRootContainer::destroy();
          return 1;
        }
```

```
CModel* pModel = pDataModel->getModel();
assert(pModel != NULL);
// create a report with the correct filename and all the species against
// time.
CReportDefinitionVector* pReports = pDataModel->getReportDefinitionList();
// create a new report definition object
CReportDefinition* pReport = pReports->createReportDefinition("Report", "Output for timecourse");
// set the task type for the report definition to timecourse
pReport->setTaskType(CCopasiTask::timeCourse);
// we don't want a table
pReport->setIsTable(false);
// the entries in the output should be seperated by a ", "
pReport->setSeparator(CCopasiReportSeparator(", "));

// we need a handle to the header and the body
// the header will display the ids of the metabolites and "time" for
// the first column
// the body will contain the actual timecourse data
std::vector<CRegisteredObjectName>* pHeader = pReport->getHeaderAddr();
std::vector<CRegisteredObjectName>* pBody = pReport->getBodyAddr();
pBody->push_back(CCopasiObjectName(pDataModel->getModel()->getCN() + ",Reference=Time"));
pBody->push_back(CRegisteredObjectName(pReport->getSeparator().getCN()));
pHeader->push_back(CCopasiStaticString("time").getCN());
pHeader->push_back(pReport->getSeparator().getCN());

size_t i, iMax = pModel->getMetabolites().size();

for (i = 0; i < iMax; ++i)
  {
    CMetab* pMetab = pModel->getMetabolites()[i];
    assert(pMetab != NULL);

    // we don't want output for FIXED metabolites right now
    if (pMetab->getStatus() != CModelEntity::FIXED)
      {
        // we want the concentration oin the output
        // alternatively, we could use "Reference=Amount" to get the
        // particle number
        pBody->push_back(pMetab->getObject(CCopasiObjectName("Reference=Concentration"))->getCN());
        // after each entry, we need a seperator
        pBody->push_back(pReport->getSeparator().getCN());

        // add the corresponding id to the header
        pHeader->push_back(CCopasiStaticString(pMetab->getSBMLId()).getCN());
        // and a seperator
        pHeader->push_back(pReport->getSeparator().getCN());
      }
  }

if (iMax > 0)
  {
    // delete the last separator
    // since we don't need one after the last element on each line
    if ((*pBody->rbegin()) == pReport->getSeparator().getCN())
      {
        pBody->erase(--pBody->end());
      }

    if ((*pHeader->rbegin()) == pReport->getSeparator().getCN())
```

```
      {
        pHeader->erase(--pHeader->end());
      }
  }

// get the task list
CCopasiVectorN< CCopasiTask > & TaskList = * pDataModel->getTaskList();

// get the trajectory task object
CTrajectoryTask* pTrajectoryTask = dynamic_cast<CTrajectoryTask*>(TaskList["Time- ↩
    Course"]);

// if there isn't one
if (pTrajectoryTask == NULL)
  {
    // create a new one
    pTrajectoryTask = new CTrajectoryTask();
    // remove any existing trajectory task just to be sure since in
    // theory only the cast might have failed above
    TaskList.remove("Time-Course");

    // add the new time course task to the task list
    TaskList.add(pTrajectoryTask, true);
  }

// run a deterministic time course
pTrajectoryTask->setMethodType(CCopasiMethod::deterministic);

// pass a pointer of the model to the problem
pTrajectoryTask->getProblem()->setModel(pDataModel->getModel());

// actiavate the task so that it will be run when the model is saved
// and passed to CopasiSE
pTrajectoryTask->setScheduled(true);

// set the report for the task
pTrajectoryTask->getReport().setReportDefinition(pReport);
// set the output filename
pTrajectoryTask->getReport().setTarget("example3.txt");
// don't append output if the file exists, but overwrite the file
pTrajectoryTask->getReport().setAppend(false);

// get the problem for the task to set some parameters
CTrajectoryProblem* pProblem = dynamic_cast<CTrajectoryProblem*>(pTrajectoryTask-> ↩
    getProblem());

// simulate 100 steps
pProblem->setStepNumber(100);
// start at time 0
pDataModel->getModel()->setInitialTime(0.0);
// simulate a duration of 10 time units
pProblem->setDuration(10);
// tell the problem to actually generate time series data
pProblem->setTimeSeriesRequested(true);

// set some parameters for the LSODA method through the method
CTrajectoryMethod* pMethod = dynamic_cast<CTrajectoryMethod*>(pTrajectoryTask-> ↩
    getMethod());

CCopasiParameter* pParameter = pMethod->getParameter("Absolute Tolerance");
assert(pParameter != NULL);
pParameter->setValue(1.0e-12);
```

```
    try
      {
        // initialize the trajectory task
        // we want complete output (HEADER, BODY and FOOTER)
        //
        // The output has to be set to OUTPUT_UI, otherwise the time series will not be
        // kept in memory and some of the assert further down will fail
        // If it is OK that the output is only written to file, the output type can
        // be set to OUTPUT_SE
        pTrajectoryTask->initialize(CCopasiTask::OUTPUT_UI, pDataModel, NULL);
        // now we run the actual trajectory
        pTrajectoryTask->process(true);
      }
    catch (...)
      {
        std::cerr << "Error. Running the time course simulation failed." << std::endl;

        // check if there are additional error messages
        if (CCopasiMessage::size() > 0)
          {
            // print the messages in chronological order
            std::cerr << CCopasiMessage::getAllMessageText(true);
          }

        CCopasiRootContainer::destroy();
        return 1;
      }

    // restore the state of the trajectory
    pTrajectoryTask->restore();

    // look at the timeseries
    const CTimeSeries* pTimeSeries = &pTrajectoryTask->getTimeSeries();
    // we simulated 100 steps, including the initial state, this should be
    // 101 step in the timeseries
    assert(pTimeSeries->getRecordedSteps() == 101);
    std::cout << "The time series consists of " << pTimeSeries->getRecordedSteps() << "." ↩
         << std::endl;
    std::cout << "Each step contains " << pTimeSeries->getNumVariables() << " variables." ↩
         << std::endl;
    std::cout << "The final state is: " << std::endl;
    iMax = pTimeSeries->getNumVariables();
    size_t lastIndex = pTimeSeries->getRecordedSteps() - 1;

    for (i = 0; i < iMax; ++i)
      {
        // here we get the particle number (at least for the species)
        // the unit of the other variables may not be particle numbers
        // the concentration data can be acquired with getConcentrationData
        std::cout << pTimeSeries->getTitle(i) << ": " << pTimeSeries->getData(lastIndex,  ↩
            i) << std::endl;
      }

  }
else
  {
    std::cerr << "Usage: example3 SBMLFILE" << std::endl;
    CCopasiRootContainer::destroy();
    return 1;
  }
```

```
  // clean up the library
  CCopasiRootContainer::destroy();
}
```

1.9.4 Running a scan over a timecourse simulation

This example demonstrates importing an SBML model from a string and running a parameter scan over a stochastic time course.

The example also shows how a report can be defined from the backend in order to get output to a file.

```
// Begin CVS Header
//   $Source: /fs/turing/cvs/copasi_dev/copasi/bindings/cpp_examples/example4/example4.cpp, ↩
    v $
//   $Revision: 1.2.6.2 $
//   $Name: Build-33 $
//   $Author: shoops $
//   $Date: 2011/03/30 16:00:40 $
// End CVS Header

// Copyright (C) 2011 - 2010 by Pedro Mendes, Virginia Tech Intellectual
// Properties, Inc., University of Heidelberg, and The University
// of Manchester.
// All rights reserved.

// Copyright (C) 2008 by Pedro Mendes, Virginia Tech Intellectual
// Properties, Inc., EML Research, gGmbH, University of Heidelberg,
// and The University of Manchester.
// All rights reserved.

/**
 * This is an example on how to import an sbml model from a string
 * create a report for a time course simulation
 * and run a scan for a stochastic time course simulation
 */

#include <iostream>
#include <string>

#define COPASI_MAIN

#include "copasi/copasi.h"
#include "copasi/report/CCopasiRootContainer.h"
#include "copasi/CopasiDataModel/CCopasiDataModel.h"
#include "copasi/model/CModel.h"
#include "copasi/model/CMetab.h"
#include "copasi/report/CReport.h"
#include "copasi/report/CReportDefinition.h"
#include "copasi/report/CReportDefinitionVector.h"
#include "copasi/trajectory/CTrajectoryTask.h"
#include "copasi/trajectory/CTrajectoryMethod.h"
#include "copasi/trajectory/CTrajectoryProblem.h"
#include "copasi/scan/CScanTask.h"
#include "copasi/scan/CScanMethod.h"
#include "copasi/scan/CScanProblem.h"

extern const char* MODEL_STRING;

int main()
{
```

```
// initialize the backend library
CCopasiRootContainer::init(0, NULL);
assert(CCopasiRootContainer::getRoot() != NULL);
// create a new datamodel
CCopasiDataModel* pDataModel = CCopasiRootContainer::addDatamodel();
assert(CCopasiRootContainer::getDatamodelList()->size() == 1);

// the only argument to the main routine should be the name of an SBML file
try
  {
    // load the model without progress report
    pDataModel->importSBMLFromString(MODEL_STRING, NULL);
  }
catch (...)
  {
    std::cerr << "Error while importing the model from the given string." << std::endl;
    CCopasiRootContainer::destroy();
    return 1;
  }

CModel* pModel = pDataModel->getModel();
assert(pModel != NULL);
// create a report with the correct filename and all the species against
// time.
CReportDefinitionVector* pReports = pDataModel->getReportDefinitionList();
// create a new report definition object
CReportDefinition* pReport = pReports->createReportDefinition("Report", "Output for  ↩
    timecourse");
// set the task type for the report definition to timecourse
pReport->setTaskType(CCopasiTask::timeCourse);
// we don't want a table
pReport->setIsTable(false);
// the entries in the output should be seperated by a ", "
pReport->setSeparator(CCopasiReportSeparator(", "));

// we need a handle to the header and the body
// the header will display the ids of the metabolites and "time" for
// the first column
// the body will contain the actual timecourse data
std::vector<CRegisteredObjectName>* pHeader = pReport->getHeaderAddr();
std::vector<CRegisteredObjectName>* pBody = pReport->getBodyAddr();
pBody->push_back(CCopasiObjectName(pDataModel->getModel()->getCN() + ",Reference=Time"));
pBody->push_back(CRegisteredObjectName(pReport->getSeparator().getCN()));
pHeader->push_back(CCopasiStaticString("time").getCN());
pHeader->push_back(pReport->getSeparator().getCN());

size_t i, iMax = pModel->getMetabolites().size();

for (i = 0; i < iMax; ++i)
  {
    CMetab* pMetab = pModel->getMetabolites()[i];
    assert(pMetab != NULL);

    // we don't want output for FIXED metabolites right now
    if (pMetab->getStatus() != CModelEntity::FIXED)
      {
        // we want the concentration oin the output
        // alternatively, we could use "Reference=Amount" to get the
        // particle number
        pBody->push_back(pMetab->getObject(CCopasiObjectName("Reference=Concentration")) ↩
            ->getCN());
        // after each entry, we need a seperator
```

```
        pBody->push_back(pReport->getSeparator().getCN());

        // add the corresponding id to the header
        pHeader->push_back(CCopasiStaticString(pMetab->getSBMLId()).getCN());
        // and a seperator
        pHeader->push_back(pReport->getSeparator().getCN());
      }
  }

if (iMax > 0)
  {
    // delete the last separator
    // since we don't need one after the last element on each line
    if ((*pBody->rbegin()) == pReport->getSeparator().getCN())
      {
        pBody->erase(--pBody->end());
      }

    if ((*pHeader->rbegin()) == pReport->getSeparator().getCN())
      {
        pHeader->erase(--pHeader->end());
      }
  }

// get the task list
CCopasiVectorN< CCopasiTask > & TaskList = * pDataModel->getTaskList();

// get the trajectory task object
CTrajectoryTask* pTrajectoryTask = dynamic_cast<CTrajectoryTask*>(TaskList["Time-Course ↩
    "]);

// if there isn't one
if (pTrajectoryTask == NULL)
  {
    // create a new one
    pTrajectoryTask = new CTrajectoryTask();
    // remove any existing trajectory task just to be sure since in
    // theory only the cast might have failed above
    TaskList.remove("Time-Course");

    // add the new time course task to the task list
    TaskList.add(pTrajectoryTask, true);
  }

// run a stochastic time course
pTrajectoryTask->setMethodType(CCopasiMethod::stochastic);

// pass a pointer of the model to the problem
pTrajectoryTask->getProblem()->setModel(pDataModel->getModel());

// we don't want the trajectory task to run by itself, but we want to
// run it from a scan, so we deactivate the standalone trajectory task
pTrajectoryTask->setScheduled(false);

// get the problem for the task to set some parameters
CTrajectoryProblem* pProblem = dynamic_cast<CTrajectoryProblem*>(pTrajectoryTask-> ↩
    getProblem());

// simulate 100 steps
pProblem->setStepNumber(100);
// start at time 0
pDataModel->getModel()->setInitialTime(0.0);
```

```
// simulate a duration of 10 time units
pProblem->setDuration(10);
// tell the problem to actually generate time series data
pProblem->setTimeSeriesRequested(true);

// now we set up the scan
CScanTask* pScanTask = dynamic_cast<CScanTask*>(TaskList["Scan"]);

if (pScanTask == NULL)
  {
    // create a new scan task
    pScanTask = new CScanTask();
    // just to be on the save side, delete any existing scan task
    TaskList.remove("Scan");
    // add the new scan task
    TaskList.add(pScanTask, true);
  }

// get the problem
CScanProblem* pScanProblem = dynamic_cast<CScanProblem*>(pScanTask->getProblem());
assert(pScanProblem != NULL);

// set the model for the problem
pScanProblem->setModel(pDataModel->getModel());

// Activate the task so that is is run
// if the model is saved and passed to CopasiSE
pScanTask->setScheduled(true);

// set the report for the task
pScanTask->getReport().setReportDefinition(pReport);

// set the output file for the report
pScanTask->getReport().setTarget("example4.txt");
// don't append to an existing file, but overwrite
pScanTask->getReport().setAppend(false);

// tell the scan that we want to make a scan over a trajectory task
pScanProblem->setSubtask(CCopasiTask::timeCourse);

// we just want to run the timecourse task a number of times, so we
// create a repeat item with 100 repeats
pScanProblem->createScanItem(CScanProblem::SCAN_REPEAT, 100);

// we want the output from the trajectory task
pScanProblem->setOutputInSubtask(true);

// we don't want to set the initial conditions of the model to the end
// state of the last run
pScanProblem->setAdjustInitialConditions(false);


try
  {
    // initialize the trajectory task
    // we want complete output (HEADER, BODY and FOOTER)
    pScanTask->initialize(CCopasiTask::OUTPUT_SE, pDataModel, NULL);
    // now we run the actual trajectory
    pScanTask->process(true);
  }
catch (...)
  {
```

```
      std::cerr << "Error. Running the scan failed." << std::endl;

      // check if there are additional error messages
      if (CCopasiMessage::size() > 0)
        {
          // print the messages in chronological order
          std::cerr << CCopasiMessage::getAllMessageText(true);
        }

      CCopasiRootContainer::destroy();
      return 1;
    }

  // restore the state of the trajectory
  pScanTask->restore();

  // clean up the library
  CCopasiRootContainer::destroy();
}


const char* MODEL_STRING =
  "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n"
  "<!-- Created by COPASI version 4.4.29 (Debug) on 2009-03-05 14:41 with libSBML version  ↩
      3.3.0. -->\n"
  "<sbml xmlns=\"http://www.sbml.org/sbml/level2/version3\" level=\"2\" version=\"3\">\n"
  "  <model metaid=\"COPASI1\" id=\"Model_1\" name=\"New Model\">\n"
  "    <listOfUnitDefinitions>\n"
  "      <unitDefinition id=\"volume\">\n"
  "        <listOfUnits>\n"
  "          <unit kind=\"litre\" scale=\"-6\"/>\n"
  "        </listOfUnits>\n"
  "      </unitDefinition>\n"
  "      <unitDefinition id=\"substance\">\n"
  "        <listOfUnits>\n"
  "          <unit kind=\"mole\" scale=\"-9\"/>\n"
  "        </listOfUnits>\n"
  "      </unitDefinition>\n"
  "    </listOfUnitDefinitions>\n"
  "    <listOfCompartments>\n"
  "      <compartment id=\"compartment_1\" name=\"compartment\" size=\"1\"/>\n"
  "    </listOfCompartments>\n"
  "    <listOfSpecies>\n"
  "      <species metaid=\"COPASI2\" id=\"species_1\" name=\"A\" compartment=\" ↩
      compartment_1\" initialConcentration=\"1e-10\">\n"
  "      </species>\n"
  "      <species metaid=\"COPASI3\" id=\"species_2\" name=\"B\" compartment=\" ↩
      compartment_1\" initialConcentration=\"0\">\n"
  "      </species>\n"
  "      <species metaid=\"COPASI4\" id=\"species_3\" name=\"C\" compartment=\" ↩
      compartment_1\" initialConcentration=\"0\">\n"
  "      </species>\n"
  "    </listOfSpecies>\n"
  "    <listOfReactions>\n"
  "      <reaction id=\"reaction_1\" name=\"reaction\" reversible=\"false\">\n"
  "        <listOfReactants>\n"
  "          <speciesReference species=\"species_1\"/>\n"
  "        </listOfReactants>\n"
  "        <listOfProducts>\n"
  "          <speciesReference species=\"species_2\"/>\n"
  "        </listOfProducts>\n"
  "        <kineticLaw>\n"
```

```
  "          <math xmlns=\"http://www.w3.org/1998/Math/MathML\">\n"
  "            <apply>\n"
  "              <times/>\n"
  "              <ci> compartment_1 </ci>\n"
  "              <ci> k1 </ci>\n"
  "              <ci> species_1 </ci>\n"
  "            </apply>\n"
  "          </math>\n"
  "          <listOfParameters>\n"
  "            <parameter id=\"k1\" value=\"0.1\"/>\n"
  "          </listOfParameters>\n"
  "        </kineticLaw>\n"
  "      </reaction>\n"
  "      <reaction id=\"reaction_2\" name=\"reaction_1\" reversible=\"false\">\n"
  "        <listOfReactants>\n"
  "          <speciesReference species=\"species_2\"/>\n"
  "        </listOfReactants>\n"
  "        <listOfProducts>\n"
  "          <speciesReference species=\"species_3\"/>\n"
  "        </listOfProducts>\n"
  "        <kineticLaw>\n"
  "          <math xmlns=\"http://www.w3.org/1998/Math/MathML\">\n"
  "            <apply>\n"
  "              <times/>\n"
  "              <ci> compartment_1 </ci>\n"
  "              <ci> k1 </ci>\n"
  "              <ci> species_2 </ci>\n"
  "            </apply>\n"
  "          </math>\n"
  "          <listOfParameters>\n"
  "            <parameter id=\"k1\" value=\"0.1\"/>\n"
  "          </listOfParameters>\n"
  "        </kineticLaw>\n"
  "      </reaction>\n"
  "    </listOfReactions>\n"
  "  </model>\n"
  "</sbml>\n";
```

1.9.5 Running an optimization task

This example shows how to run an optimization task and how to get the results of the optimization after running it.

```
// Begin CVS Header
//   $Source: /fs/turing/cvs/copasi_dev/copasi/bindings/cpp_examples/example5/example5.cpp, ↩
    v $
//   $Revision: 1.4.4.3 $
//   $Name: Build-33 $
//   $Author: shoops $
//   $Date: 2011/11/23 15:36:01 $
// End CVS Header

// Copyright (C) 2011 - 2010 by Pedro Mendes, Virginia Tech Intellectual
// Properties, Inc., University of Heidelberg, and The University
// of Manchester.
// All rights reserved.

// Copyright (C) 2008 by Pedro Mendes, Virginia Tech Intellectual
// Properties, Inc., EML Research, gGmbH, University of Heidelberg,
// and The University of Manchester.
```

```
// All rights reserved.

/**
 * This is an example on how to run an optimization task.
 * And how to access the result of an optimization.
 */

#include <iostream>
#include <string>
#include <set>
#include <math.h>
#include <assert.h>
#include <stdlib.h>

#define COPASI_MAIN

#include "copasi/copasi.h"
#include "copasi/commandline/COptions.h"
#include "copasi/report/CCopasiContainer.h"
#include "copasi/CopasiDataModel/CCopasiDataModel.h"
#include "copasi/model/CModel.h"
#include "copasi/model/CModelValue.h"
#include "copasi/report/CReport.h"
#include "copasi/report/CReportDefinition.h"
#include "copasi/report/CReportDefinitionVector.h"
#include "copasi/trajectory/CTrajectoryTask.h"
#include "copasi/trajectory/CTrajectoryMethod.h"
#include "copasi/trajectory/CTrajectoryProblem.h"
#include "copasi/optimization/COptTask.h"
#include "copasi/optimization/COptMethod.h"
#include "copasi/optimization/COptProblem.h"
#include "copasi/optimization/COptItem.h"
#include "copasi/report/CCopasiRootContainer.h"

int main()
{
  // initialize the backend library
  CCopasiRootContainer::init(0, NULL);
  assert(CCopasiRootContainer::getRoot() != NULL);
  // create a new datamodel
  CCopasiDataModel* pDataModel = CCopasiRootContainer::addDatamodel();
  assert(CCopasiRootContainer::getDatamodelList()->size() == 1);
  CModel* pModel = pDataModel->getModel();
  assert(pModel != NULL);
  pModel->setVolumeUnit(CModel::fl);
  pModel->setTimeUnit(CModel::s);
  pModel->setQuantityUnit(CModel::fMol);
  CModelValue* pFixedModelValue = pModel->createModelValue("F");
  assert(pFixedModelValue != NULL);
  pFixedModelValue->setStatus(CModelEntity::FIXED);
  pFixedModelValue->setInitialValue(3.0);
  CModelValue* pVariableModelValue = pModel->createModelValue("V");
  assert(pVariableModelValue != NULL);
  pVariableModelValue->setStatus(CModelEntity::ASSIGNMENT);
  // we create a very simple assignment that is easy on the optimization
  // a parabole with the minimum at x=6 should do just fine
  std::string s = pFixedModelValue->getObject(CCopasiObjectName("Reference=Value"))->getCN ↩
      ();
  s = "(<" + s + "> - 6.0)^2";
  pVariableModelValue->setExpression(s);
  // now we compile the model and tell COPASI which values have changed so
  // that COPASI can update the values that depend on those
```

```
std::set<const CCopasiObject*> changedObjects;
changedObjects.insert(pFixedModelValue->getObject(CCopasiObjectName("Reference= ↩
    InitialValue")));
changedObjects.insert(pVariableModelValue->getObject(CCopasiObjectName("Reference= ↩
    InitialValue")));
// finally compile the model
// compile needs to be done before updating all initial values for
// the model with the refresh sequence
pModel->compileIfNecessary(NULL);

// now that we are done building the model, we have to make sure all
// initial values are updated according to their dependencies
std::vector<Refresh*> refreshes = pModel->buildInitialRefreshSequence(changedObjects);
std::vector<Refresh*>::iterator it2 = refreshes.begin(), endit2 = refreshes.end();

while (it2 != endit2)
  {
    // call each refresh
    (**it2)();
    ++it2;
  }

// now we set up the optimization

// we want to do an optimization for the time course
// so we have to set up the time course task first
// get the task list
CCopasiVectorN< CCopasiTask > & TaskList = * pDataModel->getTaskList();

// get the optimization task
COptTask* pOptTask = dynamic_cast<COptTask*>(TaskList["Optimization"]);
assert(pOptTask != NULL);
// we want to use Levenberg-Marquardt as the optimization method
pOptTask->setMethodType(CCopasiMethod::LevenbergMarquardt);

// next we need to set subtask type on the problem
COptProblem* pOptProblem = dynamic_cast<COptProblem*>(pOptTask->getProblem());
assert(pOptProblem != NULL);
pOptProblem->setModel(pModel);
pOptProblem->setSubtaskType(CCopasiTask::timeCourse);

CTrajectoryTask* pTimeCourseTask = dynamic_cast<CTrajectoryTask*>(TaskList["Time-Course ↩
    "]);
assert(pTimeCourseTask != NULL);
// since for this example it really doesn't matter how long we run the time course
// we run for 1 second and calculate 10 steps
// run a deterministic time course
pTimeCourseTask->setMethodType(CCopasiMethod::deterministic);

// pass a pointer of the model to the problem
pTimeCourseTask->getProblem()->setModel(pModel);

// get the problem for the task to set some parameters
CTrajectoryProblem* pProblem = dynamic_cast<CTrajectoryProblem*>(pTimeCourseTask-> ↩
    getProblem());
assert(pProblem != NULL);

// simulate 10 steps
pProblem->setStepNumber(10);
// start at time 0
pModel->setInitialTime(0.0);
// simulate a duration of 1 time units
```

```
pProblem->setDuration(1);
// tell the problem to actually generate time series data
pProblem->setTimeSeriesRequested(true);
// we create the objective function
// we want to minimize the value of the variable model value at the end of
// the simulation
// the objective function is normally minimized
std::string objectiveFunction = pVariableModelValue->getObject(CCopasiObjectName(" ↩
    Reference=Value"))->getCN();
// we need to put the angled brackets around the common name of the object
objectiveFunction = "<" + objectiveFunction + ">";
// now we set the objective function in the problem
pOptProblem->setObjectiveFunction(objectiveFunction);

// now we create the optimization items
// i.e. the model elements that have to be changed during the optimization
// in order to get to the optimal solution
COptItem* pOptItem = &pOptProblem->addOptItem(CCopasiObjectName(pFixedModelValue-> ↩
    getObject(CCopasiObjectName("Reference=InitialValue"))->getCN()));
// we want to change the fixed model value from -100 to +100 with a start
// value of 50
pOptItem->setStartValue(50.0);
pOptItem->setLowerBound(CCopasiObjectName("-100"));
pOptItem->setUpperBound(CCopasiObjectName("100"));

// now we set some parameters on the method
// these parameters are specific to the method type we set above
// (in this case Levenberg-Marquardt)
COptMethod* pOptMethod = dynamic_cast<COptMethod*>(pOptTask->getMethod());
assert(pOptMethod != NULL);

// now we set some method parameters for the optimization method
// iteration limit
CCopasiParameter* pParameter = pOptMethod->getParameter("Iteration Limit");
assert(pParameter != NULL);
pParameter->setValue((C_INT32) 2000);
// tolerance
pParameter = pOptMethod->getParameter("Tolerance");
assert(pParameter != NULL);
pParameter->setValue(1.0e-5);

// create a report with the correct filename and all the species against
// time.
CReportDefinitionVector* pReports = pDataModel->getReportDefinitionList();
// create a new report definition object
CReportDefinition* pReport = pReports->createReportDefinition("Report", "Output for  ↩
    optimization");
// set the task type for the report definition to timecourse
pReport->setTaskType(CCopasiTask::optimization);
// we don't want a table
pReport->setIsTable(false);
// the entries in the output should be separated by a ", "
pReport->setSeparator(CCopasiReportSeparator(", "));

// we need a handle to the header and the body
// the header will display the ids of the metabolites and "time" for
// the first column
// the body will contain the actual timecourse data
std::vector<CRegisteredObjectName>* pHeader = pReport->getHeaderAddr();
std::vector<CRegisteredObjectName>* pBody = pReport->getBodyAddr();

// in the report header we write two strings and a separator
```

```
pHeader->push_back(CRegisteredObjectName(CCopasiStaticString("best value of objective  ↩
    function").getCN()));
pHeader->push_back(CRegisteredObjectName(pReport->getSeparator().getCN()));
pHeader->push_back(CRegisteredObjectName(CCopasiStaticString("initial value of F").getCN ↩
    ()));
// in the report body we write the best value of the objective function and
// the initial value of the fixed parameter separated by a comma
pBody->push_back(CRegisteredObjectName(pOptProblem->getObject(CCopasiObjectName(" ↩
    Reference=Best Value"))->getCN()));
pBody->push_back(CRegisteredObjectName(pReport->getSeparator().getCN()));
pBody->push_back(CRegisteredObjectName(pFixedModelValue->getObject(CCopasiObjectName(" ↩
    Reference=InitialValue"))->getCN()));

// set the report for the task
pOptTask->getReport().setReportDefinition(pReport);
// set the output filename
pOptTask->getReport().setTarget("example5.txt");
// don't append output if the file exists, but overwrite the file
pOptTask->getReport().setAppend(false);

bool result = false;

try
  {
    // initialize the trajectory task
    // we want complete output (HEADER, BODY and FOOTER)
    pOptTask->initialize(CCopasiTask::OUTPUT_SE, pDataModel, NULL);
    // now we run the actual trajectory
    result = pOptTask->process(true);
  }
catch (...)
  {
    std::cerr << "Error. Running the optimization failed." << std::endl;

    // check if there are additional error messages
    if (CCopasiMessage::size() > 0)
      {
        // print the messages in chronological order
        std::cerr << CCopasiMessage::getAllMessageText(true);
      }

    // clean up the library
    CCopasiRootContainer::destroy();
    exit(1);
  }

if (!result)
  {
    std::cerr << "Running the optimization failed." << std::endl;

    // clean up the library
    CCopasiRootContainer::destroy();
    exit(1);
  }

// restore the state of the trajectory
pOptTask->restore();
// now we check if the optimization actually got the correct result
// the best value it should have is 0 and the best parameter value for
// that result should be 6 for the initial value of the fixed parameter
double bestValue = pOptProblem->getSolutionValue();
assert(fabs(bestValue) < 1e-3);
```

```
  // we should only have one solution variable since we only have one
  // optimization item
  assert(pOptProblem->getSolutionVariables().size() == 1);
  double solution = pOptProblem->getSolutionVariables()[0];
  assert(fabs((solution - 6.0) / 6.0) < 1e-3);

  // clean up the library
  CCopasiRootContainer::destroy();
}
```

1.9.6 Doing a parameter fit

This example shows how to fit parameters using the COPASI API.

```
// Begin CVS Header
//   $Source: /fs/turing/cvs/copasi_dev/copasi/bindings/cpp_examples/example6/example6.cpp, ↩
    v $
//   $Revision: 1.4.4.3 $
//   $Name: Build-33 $
//   $Author: gauges $
//   $Date: 2011/09/14 13:37:46 $
// End CVS Header

// Copyright (C) 2011 - 2010 by Pedro Mendes, Virginia Tech Intellectual
// Properties, Inc., University of Heidelberg, and The University
// of Manchester.
// All rights reserved.

// Copyright (C) 2008 by Pedro Mendes, Virginia Tech Intellectual
// Properties, Inc., EML Research, gGmbH, University of Heidelberg,
// and The University of Manchester.
// All rights reserved.

/**
 * This is an example on how to run an parameter fitting task.
 * The example creates a simple model and runs a time course simulation on it.
 * The timecourse data is written to file with some noise added to it.
 * This data is used to fit the original parameters.
 */

#include <iostream>
#include <fstream>
#include <string>
#include <set>
#include <math.h>
#include <assert.h>
#include <stdlib.h>

#define COPASI_MAIN

#include "copasi/copasi.h"
#include "copasi/commandline/COptions.h"
#include "copasi/report/CCopasiContainer.h"
#include "copasi/CopasiDataModel/CCopasiDataModel.h"
#include "copasi/model/CModel.h"
#include "copasi/model/CModelValue.h"
#include "copasi/trajectory/CTrajectoryTask.h"
#include "copasi/trajectory/CTrajectoryMethod.h"
#include "copasi/trajectory/CTrajectoryProblem.h"
```

```
#include "copasi/parameterFitting/CFitTask.h"
#include "copasi/parameterFitting/CFitMethod.h"
#include "copasi/parameterFitting/CFitProblem.h"
#include "copasi/parameterFitting/CFitItem.h"
#include "copasi/parameterFitting/CExperimentSet.h"
#include "copasi/parameterFitting/CExperiment.h"
#include "copasi/parameterFitting/CExperimentObjectMap.h"
#include "copasi/report/CKeyFactory.h"
#include "copasi/report/CCopasiRootContainer.h"

extern const char* MODEL_STRING;

int main()
{
  // initialize the backend library
  CCopasiRootContainer::init(0, NULL);
  assert(CCopasiRootContainer::getRoot() != NULL);

  // create a new datamodel
  CCopasiDataModel* pDataModel = CCopasiRootContainer::addDatamodel();
  assert(CCopasiRootContainer::getDatamodelList()->size() == 1);

  // first we load a simple model
  try
    {
      // load the model
      pDataModel->importSBMLFromString(MODEL_STRING);
      // we clear the message stack to get rid of all error messages
      // this is important because otherwise the initialization of the
      // fit task will fail
      CCopasiMessage::clearDeque();
    }
  catch (...)
    {
      std::cerr <<  "Error while importing the model." << std::endl;
      exit(1);
    }

  // now we need to run some time course simulation to get data to fit
  // against

  // get the trajectory task object
  CCopasiVectorN< CCopasiTask > & TaskList = * pDataModel->getTaskList();

  // get the optimization task
  CTrajectoryTask* pTrajectoryTask = dynamic_cast<CTrajectoryTask*>(TaskList["Time-Course ↩
      "]);
  assert(pTrajectoryTask != NULL);

  // if there isn't one
  if (pTrajectoryTask == NULL)
    {
      // create a new one
      pTrajectoryTask = new CTrajectoryTask();

      // add the new time course task to the task list
      pDataModel->getTaskList()->add(pTrajectoryTask);
    }

  // run a deterministic time course
  pTrajectoryTask->setMethodType(CCopasiMethod::deterministic);
```

```
// pass a pointer of the model to the problem
pTrajectoryTask->getProblem()->setModel(pDataModel->getModel());

// activate the task so that it will be run when the model is saved
// and passed to CopasiSE
pTrajectoryTask->setScheduled(true);

// get the problem for the task to set some parameters
CTrajectoryProblem* pProblem = dynamic_cast<CTrajectoryProblem*>(pTrajectoryTask-> ↩
    getProblem());

// simulate 4000 steps
pProblem->setStepNumber(4000);
// start at time 0
pDataModel->getModel()->setInitialTime(0.0);
// simulate a duration of 400 time units
pProblem->setDuration(400);
// tell the problem to actually generate time series data
pProblem->setTimeSeriesRequested(true);

bool result = true;

try
  {
    // now we run the actual trajectory
    //
    // The output has to be set to OUTPUT_UI, otherwise the time series will not be
    // kept in memory and some of the assert further down will fail
    // If it is OK that the output is only written to file, the output type can
    // be set to OUTPUT_SE
    pTrajectoryTask->initialize(CCopasiTask::OUTPUT_UI, pDataModel, NULL);
    result = pTrajectoryTask->process(true);
  }
catch (...)
  {
    std::cerr <<  "Error. Running the time course simulation failed." << std::endl;

    // check if there are additional error messages
    if (CCopasiMessage::size() > 0)
      {
        // print the messages in chronological order
        std::cerr << CCopasiMessage::getAllMessageText(true) << std::endl;
      }

    // clean up the library
    CCopasiRootContainer::destroy();
    exit(1);
  }

if (result == false)
  {
    std::cerr <<  "An error occured while running the time course simulation." << std:: ↩
        endl;

    // check if there are additional error messages
    if (CCopasiMessage::size() > 0)
      {
        // print the messages in chronological order
        std::cerr << CCopasiMessage::getAllMessageText(true) << std::endl;
      }

    // clean up the library
```

```
    CCopasiRootContainer::destroy();
    exit(1);
  }

// restore the trajectory task state
pTrajectoryTask->restore();

// we write the data to a file and add some noise to it
// This is necessary since COPASI can only read experimental data from
// file.
const CTimeSeries* pTimeSeries = &pTrajectoryTask->getTimeSeries();
// we simulated 4000 steps, including the initial state, this should be
// 4001 step in the timeseries
assert(pTimeSeries->getRecordedSteps() == 4001);
size_t i, iMax = pTimeSeries->getNumVariables();
// there should be four variables, the three metabolites and time
assert(iMax == 5);
size_t lastIndex = pTimeSeries->getRecordedSteps() - 1;
// open the file
// we need to remember in which order the variables are written to file
// since we need to specify this later in the parameter fitting task
std::set<size_t> indexSet;
std::vector<CMetab*> metabVector;

// write the header
// the first variable in a time series is a always time, for the rest
// of the variables, we use the SBML id in the header
double random = 0.0;

try
  {
    std::ofstream os("fakedata_example6.txt", std::ios_base::out | std::ios_base::trunc);
    os << ("# time ");

    for (i = 1; i < iMax; ++i)
      {
        std::string key = pTimeSeries->getKey(i);
        CCopasiObject* pObject = CCopasiRootContainer::getKeyFactory()->get(key);
        assert(pObject != NULL);

        // only write header data or metabolites
        if (dynamic_cast<const CMetab*>(pObject))
          {
            os << ", ";
            os << pTimeSeries->getSBMLId(i, pDataModel);
            CMetab* pM = dynamic_cast<CMetab*>(pObject);
            indexSet.insert(i);
            metabVector.push_back(pM);
          }
      }

    os << std::endl;
    double data = 0.0;

    for (i = 0; i < lastIndex; ++i)
      {
        size_t j;
        std::ostringstream s;

        for (j = 0; j < iMax; ++j)
          {
            // we only want to  write the data for metabolites
```

```
            // the compartment does not interest us here
            if (j == 0 || indexSet.find(j) != indexSet.end())
              {
                // write the data with some noise (+-5% max)
                random = ((double)rand()) / (double)RAND_MAX;
                data = pTimeSeries->getConcentrationData(i, j);

                // don't add noise to the time
                if (j != 0)
                  {
                    data += data * (random * 0.1 - 0.05);
                  }

                s << data;
                s << ", ";
              }
          }

        // remove the last two characters again
        os << s.str().substr(0, s.str().length() - 2);
        os << std::endl;
      }

    os.close();
  }
catch (const std::exception& e)
  {
    std::cerr << "Error. Could not write time course data to file." << std::endl;
    std::cout << e.what() << std::endl;
    exit(1);
  }

// now we change the parameter values to see if the parameter fitting
// can really find the original values
random = (double)rand() / (double)RAND_MAX * 10.0;
CReaction* pReaction = pDataModel->getModel()->getReactions()[0];
// we know that it is an irreversible mass action, so there is one
// parameter
assert(pReaction->getParameters().size() == 1);
assert(pReaction->isLocalParameter(0));
// the parameter of a irreversible mass action is called k1
pReaction->setParameterValue("k1", random);

pReaction = pDataModel->getModel()->getReactions()[1];
// we know that it is an irreversible mass action, so there is one
// parameter
assert(pReaction->getParameters().size() == 1);
assert(pReaction->isLocalParameter(0));
pReaction->setParameterValue("k1", random);

CFitTask* pFitTask = dynamic_cast<CFitTask*>(TaskList["Parameter Estimation"]);
assert(pFitTask != NULL);
// the method in a fit task is an instance of COptMethod or a subclass of
// it.
COptMethod* pFitMethod = dynamic_cast<COptMethod*>(pFitTask->getMethod());
assert(pFitMethod != NULL);
// the object must be an instance of COptMethod or a subclass thereof
// (CFitMethod)
CFitProblem* pFitProblem = dynamic_cast<CFitProblem*>(pFitTask->getProblem());
assert(pFitProblem != NULL);

CExperimentSet* pExperimentSet = dynamic_cast<CExperimentSet*>(pFitProblem->getParameter ↩
```

```
    ("Experiment Set"));
assert(pExperimentSet != NULL);

// first experiment (we only have one here)
CExperiment* pExperiment = new CExperiment(pDataModel);
assert(pExperiment != NULL);
// tell COPASI where to find the data
// reading data from string is not possible with the current C++ API
pExperiment->setFileName("fakedata_example6.txt");
// we have to tell COPASI that the data for the experiment is a komma
// separated list (the default is TAB separated)
pExperiment->setSeparator(",");
// the data start in row 1 and goes to row 4001
pExperiment->setFirstRow(1);
assert(pExperiment->getFirstRow() == 1);
pExperiment->setLastRow(4001);
assert(pExperiment->getLastRow() == 4001);
pExperiment->setHeaderRow(1);
assert(pExperiment->getHeaderRow() == 1);
pExperiment->setExperimentType(CCopasiTask::timeCourse);
assert(pExperiment->getExperimentType() == CCopasiTask::timeCourse);
pExperiment->setNumColumns(4);
assert(pExperiment->getNumColumns() == 4);
CExperimentObjectMap* pObjectMap = &pExperiment->getObjectMap();
assert(pObjectMap != NULL);
result = pObjectMap->setNumCols(4);
assert(result == true);
result = pObjectMap->setRole(0, CExperiment::time);
assert(result == true);
assert(pObjectMap->getRole(0) == CExperiment::time);

CModel* pModel = pDataModel->getModel();
assert(pModel != NULL);
const CCopasiObject* pTimeReference = pModel->getObject(CCopasiObjectName("Reference=Time ↩
    "));
assert(pTimeReference != NULL);
pObjectMap->setObjectCN(0, pTimeReference->getCN());

// now we tell COPASI which column contain the concentrations of
// metabolites and belong to dependent variables
pObjectMap->setRole(1, CExperiment::dependent);
CMetab* pMetab = metabVector[0];
assert(pMetab != NULL);
const CCopasiObject* pParticleReference = pMetab->getObject(CCopasiObjectName("Reference= ↩
    Concentration"));
assert(pParticleReference != NULL);
pObjectMap->setObjectCN(1, pParticleReference->getCN());

pObjectMap->setRole(2, CExperiment::dependent);
pMetab = metabVector[1];
assert(pMetab != NULL);
pParticleReference = pMetab->getObject(CCopasiObjectName("Reference=Concentration"));
assert(pParticleReference != NULL);
pObjectMap->setObjectCN(2, pParticleReference->getCN());

pObjectMap->setRole(3, CExperiment::dependent);
pMetab = metabVector[2];
assert(pMetab != NULL);
pParticleReference = pMetab->getObject(CCopasiObjectName("Reference=Concentration"));
assert(pParticleReference != NULL);
pObjectMap->setObjectCN(3, pParticleReference->getCN());
```

```
pExperimentSet->addExperiment(*pExperiment);
assert(pExperimentSet->getExperimentCount() == 1);
// addExperiment makes a copy, so we need to get the added experiment
// again
delete pExperiment;
pExperiment = pExperimentSet->getExperiment(0);
assert(pExperiment != NULL);

// now we have to define the two fit items for the two local parameters
// of the two reactions
pReaction = pModel->getReactions()[0];
assert(pReaction != NULL);
assert(pReaction->isLocalParameter(0) == true);
CCopasiParameter* pParameter = pReaction->getParameters().getParameter(0);
assert(pParameter != NULL);

// get the list where we have to add the fit items
CCopasiParameterGroup* pOptimizationItemGroup = dynamic_cast<CCopasiParameterGroup*>( ↩
    pFitProblem->getParameter("OptimizationItemList"));
assert(pOptimizationItemGroup != NULL);

// define a CFitItem
const CCopasiObject* pParameterReference = pParameter->getObject(CCopasiObjectName(" ↩
    Reference=Value"));
assert(pParameterReference != NULL);
CFitItem* pFitItem1 = new CFitItem(pDataModel);
pFitItem1->setObjectCN(pParameterReference->getCN());
assert(pFitItem1 != NULL);
pFitItem1->setStartValue(4.0);
pFitItem1->setLowerBound(CCopasiObjectName("0.00001"));
pFitItem1->setUpperBound(CCopasiObjectName("10"));
// add the fit item
pOptimizationItemGroup->addParameter(pFitItem1);

pReaction = pModel->getReactions()[1];
assert(pReaction != NULL);
assert(pReaction->isLocalParameter(0) == true);
pParameter = pReaction->getParameters().getParameter(0);
assert(pParameter != NULL);

// define a CFitItem
pParameterReference = pParameter->getObject(CCopasiObjectName("Reference=Value"));
assert(pParameterReference != NULL);
CFitItem* pFitItem2 = new CFitItem(pDataModel);
pFitItem2->setObjectCN(pParameterReference->getCN());
assert(pFitItem2 != NULL);
pFitItem2->setStartValue(4.0);
pFitItem2->setLowerBound(CCopasiObjectName("0.00001"));
pFitItem2->setUpperBound(CCopasiObjectName("10"));
// add the fit item
pOptimizationItemGroup->addParameter(pFitItem2);

result = true;

try
  {
    // initialize the fit task
    // we want complete output (HEADER, BODY and FOOTER)
    result = pFitTask->initialize(CCopasiTask::OUTPUT_SE, pDataModel, NULL);

    if (result == true)
      {
```

```
          // running the task for this example will probably take some time
          std::cout << "This can take some time..." << std::endl;
          result = pFitTask->process(true);
        }
    }
  catch (...)
    {
      std::cerr << "Error. Parameter fitting failed." << std::endl;

      // clean up the library
      CCopasiRootContainer::destroy();
      exit(1);
    }

  pFitTask->restore();
  assert(result == true);
  // assert that there are two optimization items
  assert(pFitProblem->getOptItemList().size() == 2);
  // the order should be the order in whih we added the items above
  COptItem* pOptItem1 = pFitProblem->getOptItemList()[0];
  COptItem* pOptItem2 = pFitProblem->getOptItemList()[1];
  // the actual results are stored in the fit problem
  assert(pFitProblem->getSolutionVariables().size() == 2);
  std::cout << "value for " << pOptItem1->getObject()->getCN() << ": " << pFitProblem-> ↩
      getSolutionVariables()[0] << std::endl;
  std::cout << "value for " << pOptItem2->getObject()->getCN() << ": " << pFitProblem-> ↩
      getSolutionVariables()[1] << std::endl;
  // depending on the noise, the fit can be quite bad, so we are a litle
  // relaxed here (we should be within 3% of the original values)
  assert((fabs(pFitProblem->getSolutionVariables()[0] - 0.03) / 0.03) < 3e-2);
  assert((fabs(pFitProblem->getSolutionVariables()[1] - 0.004) / 0.004) < 3e-2);

  // clean up the library
  CCopasiRootContainer::destroy();
}

const char* MODEL_STRING =
  "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n"
  "<!-- Created by COPASI version 4.5.30 (Debug) on 2009-03-30 08:01 with libSBML version  ↩
      3.3.2. -->\n"
  "<sbml xmlns=\"http://www.sbml.org/sbml/level2\" level=\"2\" version=\"1\">\n"
  "  <model metaid=\"COPASI1\" id=\"Model_1\" name=\"New Model\">\n"
  "    <listOfUnitDefinitions>\n"
  "      <unitDefinition id=\"volume\">\n"
  "        <listOfUnits>\n"
  "          <unit kind=\"litre\" scale=\"-3\"/>\n"
  "        </listOfUnits>\n"
  "      </unitDefinition>\n"
  "      <unitDefinition id=\"substance\">\n"
  "        <listOfUnits>\n"
  "          <unit kind=\"mole\" scale=\"-3\"/>\n"
  "        </listOfUnits>\n"
  "      </unitDefinition>\n"
  "    </listOfUnitDefinitions>\n"
  "    <listOfCompartments>\n"
  "      <compartment id=\"compartment_1\" name=\"compartment\" size=\"1\"/>\n"
  "    </listOfCompartments>\n"
  "    <listOfSpecies>\n"
  "      <species id=\"species_1\" name=\"A\" compartment=\"compartment_1\"  ↩
      initialConcentration=\"5\"/>\n"
  "      <species id=\"species_2\" name=\"B\" compartment=\"compartment_1\"  ↩
      initialConcentration=\"0\"/>\n"
```

```
  "      <species id=\"species_3\" name=\"C\" compartment=\"compartment_1\"  ↩
      initialConcentration=\"0\"/>\n"
  "    </listOfSpecies>\n"
  "    <listOfReactions>\n"
  "      <reaction id=\"reaction_1\" name=\"reaction\" reversible=\"false\">\n"
  "        <listOfReactants>\n"
  "          <speciesReference species=\"species_1\"/>\n"
  "        </listOfReactants>\n"
  "        <listOfProducts>\n"
  "          <speciesReference species=\"species_2\"/>\n"
  "        </listOfProducts>\n"
  "        <kineticLaw>\n"
  "          <math xmlns=\"http://www.w3.org/1998/Math/MathML\">\n"
  "            <apply>\n"
  "              <times/>\n"
  "              <ci> compartment_1 </ci>\n"
  "              <ci> k1 </ci>\n"
  "              <ci> species_1 </ci>\n"
  "            </apply>\n"
  "          </math>\n"
  "          <listOfParameters>\n"
  "            <parameter id=\"k1\" name=\"k1\" value=\"0.03\"/>\n"
  "          </listOfParameters>\n"
  "        </kineticLaw>\n"
  "      </reaction>\n"
  "      <reaction id=\"reaction_2\" name=\"reaction_1\" reversible=\"false\">\n"
  "        <listOfReactants>\n"
  "          <speciesReference species=\"species_2\"/>\n"
  "        </listOfReactants>\n"
  "        <listOfProducts>\n"
  "          <speciesReference species=\"species_3\"/>\n"
  "        </listOfProducts>\n"
  "        <kineticLaw>\n"
  "          <math xmlns=\"http://www.w3.org/1998/Math/MathML\">\n"
  "            <apply>\n"
  "              <times/>\n"
  "              <ci> compartment_1 </ci>\n"
  "              <ci> k1 </ci>\n"
  "              <ci> species_2 </ci>\n"
  "            </apply>\n"
  "          </math>\n"
  "          <listOfParameters>\n"
  "            <parameter id=\"k1\" name=\"k1\" value=\"0.004\"/>\n"
  "          </listOfParameters>\n"
  "        </kineticLaw>\n"
  "      </reaction>\n"
  "    </listOfReactions>\n"
  "  </model>\n"
  "</sbml>";
```

1.9.7 Creating and using function definitions

This example shows how to create a function definition with the COPASI API and how to use it as a kinetic law in a reaction.

```
// Begin CVS Header
//   $Source: /fs/turing/cvs/copasi_dev/copasi/bindings/cpp_examples/example7/example7.cpp, ↩
    v $
//   $Revision: 1.3.4.1 $
//   $Name: Build-33 $
```

```
//   $Author: shoops $
//   $Date: 2011/01/13 19:36:30 $
// End CVS Header

// Copyright (C) 2010 by Pedro Mendes, Virginia Tech Intellectual
// Properties, Inc., University of Heidelberg, and The University
// of Manchester.
// All rights reserved.

// Copyright (C) 2008 by Pedro Mendes, Virginia Tech Intellectual
// Properties, Inc., EML Research, gGmbH, University of Heidelberg,
// and The University of Manchester.
// All rights reserved.

/**
 * This is an example on how to create user defined kinetic functions with the COPASI API
 */
#include <iostream>
#include <vector>
#include <string>
#include <set>

#define COPASI_MAIN

#include "copasi/copasi.h"
#include "copasi/report/CCopasiRootContainer.h"
#include "copasi/CopasiDataModel/CCopasiDataModel.h"
#include "copasi/model/CModel.h"
#include "copasi/model/CCompartment.h"
#include "copasi/model/CMetab.h"
#include "copasi/model/CReaction.h"
#include "copasi/model/CChemEq.h"
#include "copasi/model/CModelValue.h"
#include "copasi/function/CFunctionDB.h"
#include "copasi/function/CFunction.h"
#include "copasi/function/CEvaluationTree.h"

int main()
{
  // initialize the backend library
  // since we are not interested in the arguments
  // that are passed to main, we pass 0 and NULL to
  // init
  CCopasiRootContainer::init(0, NULL);
  assert(CCopasiRootContainer::getRoot() != NULL);
  // create a new datamodel
  CCopasiDataModel* pDataModel = CCopasiRootContainer::addDatamodel();
  assert(CCopasiRootContainer::getDatamodelList()->size() == 1);
  // get the model from the datamodel
  CModel* pModel = pDataModel->getModel();
  assert(pModel != NULL);
  // set the units for the model
  // we want seconds as the time unit
  // microliter as the volume units
  // and nanomole as the substance units
  pModel->setTimeUnit(CModel::s);
  pModel->setVolumeUnit(CModel::microl);
  pModel->setQuantityUnit(CModel::nMol);

  // we have to keep a set of all the initial values that are changed during
  // the model building process
  // They are needed after the model has been built to make sure all initial
```

```
// values are set to the correct initial value
std::set<const CCopasiObject*> changedObjects;

// create a compartment with the name cell and an initial volume of 5.0
// microliter
CCompartment* pCompartment = pModel->createCompartment("cell", 5.0);
const CCopasiObject* pObject = pCompartment->getObject(CCopasiObjectName("Reference= ↩
    InitialVolume"));
assert(pObject != NULL);
changedObjects.insert(pObject);
assert(pCompartment != NULL);
assert(pModel->getCompartments().size() == 1);
// create a new metabolite with the name S and an inital
// concentration of 10 nanomol
// the metabolite belongs to the compartment we created and is is to be
// fixed
CMetab* pS = pModel->createMetabolite("S", pCompartment->getObjectName(), 10.0, CMetab:: ↩
    FIXED);
pObject = pS->getObject(CCopasiObjectName("Reference=InitialConcentration"));
assert(pObject != NULL);
changedObjects.insert(pObject);
assert(pCompartment != NULL);
assert(pS != NULL);
assert(pModel->getMetabolites().size() == 1);
// create a second metabolite called P with an initial
// concentration of 0. This metabolite is to be changed by reactions
CMetab* pP = pModel->createMetabolite("P", pCompartment->getObjectName(), 0.0, CMetab:: ↩
    REACTIONS);
assert(pP != NULL);
pObject = pP->getObject(CCopasiObjectName("Reference=InitialConcentration"));
assert(pObject != NULL);
changedObjects.insert(pObject);
assert(pModel->getMetabolites().size() == 2);
// now we create a reaction
CReaction* pReaction = pModel->createReaction("reaction");
assert(pReaction != NULL);
assert(pModel->getReactions().size() == 1);
// reaction converts S to P
// we can set these on the chemical equation of the reaction
CChemEq* pChemEq = &pReaction->getChemEq();
// S is a substrate with stoichiometry 1
pChemEq->addMetabolite(pS->getKey(), 1.0, CChemEq::SUBSTRATE);
// P is a product with stoichiometry 1
pChemEq->addMetabolite(pP->getKey(), 1.0, CChemEq::PRODUCT);
assert(pChemEq->getSubstrates().size() == 1);
assert(pChemEq->getProducts().size() == 1);
// this reaction is to be irreversible
pReaction->setReversible(false);
assert(pReaction->isReversible() == false);

CModelValue* pMV = pModel->createModelValue("K", 42.0);
// set the status to FIXED
pMV->setStatus(CModelValue::FIXED);
assert(pMV != NULL);
pObject = pMV->getObject(CCopasiObjectName("Reference=InitialValue"));
assert(pObject != NULL);
changedObjects.insert(pObject);
assert(pModel->getModelValues().size() == 1);

// now we ned to set a kinetic law on the reaction
// for this we create a user defined function
CFunctionDB* pFunDB = CCopasiRootContainer::getFunctionList();
```

```
assert(pFunDB != NULL);

CKinFunction* pFunction = new CKinFunction("My Rate Law");

pFunDB->add(pFunction, true);
CFunction* pRateLaw = dynamic_cast<CFunction*>(pFunDB->findFunction("My Rate Law"));

assert(pRateLaw != NULL);

// now we create the formula for the function and set it on the function
std::string formula = "(1-0.4/(EXPONENTIALE^(temp-37)))*0.00001448471257*1.4^(temp-37)* ↩
    substrate";

bool result = pFunction->setInfix(formula);
assert(result == true);
// make the function irreversible
pFunction->setReversible(TriFalse);
// the formula string should have been parsed now
// and COPASI should have determined that the formula string contained 2 parameters (temp ↩
     and substrate)
CFunctionParameters& variables = pFunction->getVariables();
// per default the usage of those parameters will be set to VARIABLE
size_t index = pFunction->getVariableIndex("temp");
assert(index != C_INVALID_INDEX);
CFunctionParameter* pParam = variables[index];
assert(pParam->getUsage() == CFunctionParameter::VARIABLE);
// This is correct for temp, but substrate should get the usage SUBSTRATE in order
// for us to use the function with the reaction created above
// So we need to set the usage for "substrate" manually
index = pFunction->getVariableIndex("substrate");
assert(index != C_INVALID_INDEX);
pParam = variables[index];
pParam->setUsage(CFunctionParameter::SUBSTRATE);

// set the rate law for the reaction
pReaction->setFunction(pFunction);
assert(pReaction->getFunction() != NULL);

// COPASI also needs to know what object it has to assocuiate with the individual  ↩
    function parameters
// In our case we need to tell COPASI that substrate is to be replaced by the substrate  ↩
    of the reaction
// and temp is to be replaced by the global parameter K
pReaction->setParameterMapping("substrate", pS->getKey());
pReaction->setParameterMapping("temp", pMV->getKey());

// finally compile the model
// compile needs to be done before updating all initial values for
// the model with the refresh sequence
pModel->compileIfNecessary(NULL);

// now that we are done building the model, we have to make sure all
// initial values are updated according to their dependencies
std::vector<Refresh*> refreshes = pModel->buildInitialRefreshSequence(changedObjects);
std::vector<Refresh*>::iterator it2 = refreshes.begin(), endit2 = refreshes.end();

while (it2 != endit2)
  {
    // call each refresh
    (**it2)();
    ++it2;
  }
```

```
  // save the model to a COPASI file
  // we save to a file named example1.cps, we don't want a progress report
  // and we want to overwrite any existing file with the same name
  // Default tasks are automatically generated and will always appear in cps
  // file unless they are explicitley deleted before saving.
  pDataModel->saveModel("example7.cps", NULL, true);

  // export the model to an SBML file
  // we save to a file named example1.xml, we want to overwrite any
  // existing file with the same name and we want SBML L2V3
  pDataModel->exportSBML("example7.xml", true, 2, 3);

  // destroy the root container once we are done
  CCopasiRootContainer::destroy();
}
```

1.9.8 Calculating the jacobian matrix of a model

This example shows how to calculate the jacobian matrix of a model at an arbitrary state.

```
// Begin CVS Header
//   $Source: /fs/turing/cvs/copasi_dev/copasi/bindings/cpp_examples/example8/example8.cpp, ↩
    v $
//   $Revision: 1.2.2.1 $
//   $Name: Build-33 $
//   $Author: shoops $
//   $Date: 2011/01/13 19:36:29 $
// End CVS Header


// Copyright (C) 2010 by Pedro Mendes, Virginia Tech Intellectual
// Properties, Inc., University of Heidelberg, and The University
// of Manchester.
// All rights reserved.

/**
 * This is an example on how to calculate and output the Jacobian matrix
 * in COPASI
 */
#include <iostream>
#include <iomanip>
#include <vector>

#define COPASI_MAIN
#include "copasi/copasi.h"
#include "copasi/report/CCopasiRootContainer.h"
#include "copasi/CopasiDataModel/CCopasiDataModel.h"
#include "copasi/model/CModel.h"
#include "copasi/model/CState.h"
#include "copasi/model/CModelValue.h"
#include "copasi/utilities/CMatrix.h"

const char* MODEL_STRING = \
                           "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n"
                           "<!-- Created by COPASI version 4.5.31 (Debug) on 2010-05-11  ↩
                               13:40 with libSBML version 4.1.0-b3. -->\n"
                           "<sbml xmlns=\"http://www.sbml.org/sbml/level2/version4\" level ↩
                               =\"2\" version=\"4\">\n"
```

```
"  <model metaid=\"COPASI1\" id=\"Model_1\" name=\"New Model\">\ ↩
   n"
"    <listOfUnitDefinitions>\n"
"      <unitDefinition id=\"volume\" name=\"volume\">\n"
"        <listOfUnits>\n"
"          <unit kind=\"litre\" scale=\"-3\"/>\n"
"        </listOfUnits>\n"
"      </unitDefinition>\n"
"      <unitDefinition id=\"substance\" name=\"substance\">\n"
"        <listOfUnits>\n"
"          <unit kind=\"mole\" scale=\"-3\"/>\n"
"        </listOfUnits>\n"
"      </unitDefinition>\n"
"      <unitDefinition id=\"unit_0\">\n"
"        <listOfUnits>\n"
"          <unit kind=\"second\" exponent=\"-1\"/>\n"
"        </listOfUnits>\n"
"      </unitDefinition>\n"
"    </listOfUnitDefinitions>\n"
"    <listOfCompartments>\n"
"      <compartment id=\"compartment_1\" name=\"compartment\"  ↩
   size=\"1\"/>\n"
"    </listOfCompartments>\n"
"    <listOfSpecies>\n"
"      <species metaid=\"COPASI2\" id=\"species_1\" name=\"A\"  ↩
   compartment=\"compartment_1\" initialConcentration=\"1\"/>\n"
"      <species metaid=\"COPASI3\" id=\"species_2\" name=\"B\"  ↩
   compartment=\"compartment_1\" initialConcentration=\"0\"/>\n"
"      <species metaid=\"COPASI4\" id=\"species_3\" name=\"C\"  ↩
   compartment=\"compartment_1\" initialConcentration=\"0\"/>\n"
"    </listOfSpecies>\n"
"    <listOfReactions>\n"
"      <reaction metaid=\"COPASI5\" id=\"reaction_1\" name=\" ↩
   reaction_1\" reversible=\"false\">\n"
"        <listOfReactants>\n"
"          <speciesReference species=\"species_1\"/>\n"
"        </listOfReactants>\n"
"        <listOfProducts>\n"
"          <speciesReference species=\"species_2\"/>\n"
"        </listOfProducts>\n"
"        <kineticLaw>\n"
"          <math xmlns=\"http://www.w3.org/1998/Math/MathML\">\n ↩
   "
"            <apply>\n"
"              <times/>\n"
"              <ci> compartment_1 </ci>\n"
"              <ci> k1 </ci>\n"
"              <ci> species_1 </ci>\n"
"            </apply>\n"
"          </math>\n"
"          <listOfParameters>\n"
"            <parameter id=\"k1\" name=\"k1\" value=\"0.2\"  ↩
   units=\"unit_0\"/>\n"
"          </listOfParameters>\n"
"        </kineticLaw>\n"
"      </reaction>\n"
"      <reaction metaid=\"COPASI6\" id=\"reaction_2\" name=\" ↩
   reaction_2\" reversible=\"false\">\n"
"        <listOfReactants>\n"
"          <speciesReference species=\"species_2\"/>\n"
"        </listOfReactants>\n"
"        <listOfProducts>\n"
```

```
                              "          <speciesReference species=\"species_3\"/>\n"
                              "        </listOfProducts>\n"
                              "        <kineticLaw>\n"
                              "          <math xmlns=\"http://www.w3.org/1998/Math/MathML\">\n ↩
                                  "
                              "            <apply>\n"
                              "              <times/>\n"
                              "              <ci> compartment_1 </ci>\n"
                              "              <ci> k1 </ci>\n"
                              "              <ci> species_2 </ci>\n"
                              "            </apply>\n"
                              "          </math>\n"
                              "          <listOfParameters>\n"
                              "            <parameter id=\"k1\" name=\"k1\" value=\"0.1\"  ↩
                                  units=\"unit_0\"/>\n"
                              "          </listOfParameters>\n"
                              "        </kineticLaw>\n"
                              "      </reaction>\n"
                              "    </listOfReactions>\n"
                              "  </model>\n"
                              "</sbml>\n";


int main()
{
  // initialize the backend library
  // since we are not interested in the arguments
  // that are passed to main, we pass 0 and NULL to
  // init
  CCopasiRootContainer::init(0, NULL);
  assert(CCopasiRootContainer::getRoot() != NULL);
  // create a new datamodel
  CCopasiDataModel* pDataModel = CCopasiRootContainer::addDatamodel();
  assert(pDataModel != NULL);
  assert(CCopasiRootContainer::getDatamodelList()->size() == 1);
  // next we import a simple SBML model from a string

  // clear the message queue so that we only have error messages from the import in the  ↩
      queue
  CCopasiMessage::clearDeque();
  bool result = pDataModel->importSBMLFromString(MODEL_STRING);
  // check if the import was successful
  CCopasiMessage::Type mostSevere = CCopasiMessage::getHighestSeverity();
  // if it was a filtered error, we convert it to an unfiltered type
  // the filtered error messages have the same value as the unfiltered, but they
  // have the 7th bit set which basically adds 128 to the value
  mostSevere = (CCopasiMessage::Type)((int)mostSevere & 127);

  // we assume that the import succeeded if the return value is true and
  // the most severe error message is not an error or an exception
  if (result != true &&  mostSevere < CCopasiMessage::ERROR)
    {
      std::cerr << "Sorry. Model could not be imported." << std::endl;
      return 1;
    }

  //
  // now we tell the model object to calculate the jacobian
  //
  CModel* pModel = pDataModel->getModel();
  assert(pModel != NULL);
```

```
if (pModel != NULL)
  {
    // running a task, e.g. a trajectory will automatically make sure that
    // the initial values are transferred to the current state before the calculation ↩
       begins.
    // If we use low level calculation methods like the one to calculate the jacobian, we
    // have to make sure the the initial values are applied to the state
    pModel->applyInitialValues();
    // we need an array that stores the result
    // the size of the matrix does not really matter because
    // the calculateJacobian automatically resizes it to the correct
    // size
    CMatrix<C_FLOAT64> jacobian;
    // the first parameter to the calculation function is a reference to
    // the matrix where the result is to be stored
    // the second parameter is the derivationFactor for the calculation
    // it basically represents a relative delta value for the calculation of the ↩
       derivatives
    // the third parameter termed resolution in the C++ API is currently ignores
    // so it does not matter what value you give here.
    pModel->calculateJacobian(jacobian, 1e-12, 1.0);
    // now we print the result
    // the jacobian stores the values in the order they are
    // given in the user order in the state template so it is not really straight
    // forward to find out which column/row corresponds to which species
    const CStateTemplate& stateTemplate = pModel->getStateTemplate();
    // we get a pointer to all entities
    CModelEntity** pEntities = const_cast<CStateTemplate&>(stateTemplate).getEntities();
    // and we need the user order
    const CVector<size_t>& userOrder = stateTemplate.getUserOrder();
    // from those two, we can construct an new vector that contains
    // the names of the entities in the jacobian in the order in which they appear in
    // the jacobian
    std::vector<std::string> nameVector;
    const CModelEntity* pEntity = NULL;
    CModelEntity::Status status;

    for (size_t i = 0; i < userOrder.size(); ++i)
      {
        pEntity = pEntities[userOrder[i]];
        assert(pEntity != NULL);
        // now we need to check if the entity is actually
        // determined by an ODE or a reaction
        status = pEntity->getStatus();

        if (status == CModelEntity::ODE ||
            (status == CModelEntity::REACTIONS && pEntity->isUsed()))
          {
            nameVector.push_back(pEntity->getObjectName());
          }
      }

    assert(nameVector.size() == jacobian.numRows());
    // now we print the matrix, for this we assume that no
    // entity name is longer then 5 character which is a save bet since
    // we know the model
    std::cout << "Jacobian Matrix:" << std::endl << std::endl;
    std::cout << std::setw(5) << " ";

    for (size_t i = 0; i < nameVector.size(); ++i)
      {
        std::cout << std::setw(5) << nameVector[i];
```

```
        }

      std::cout << std::endl;

      for (size_t i = 0; i < nameVector.size(); ++i)
        {
          std::cout << std::setw(5) << nameVector[i];

          for (size_t j = 0; j < nameVector.size(); ++j)
            {
              std::cout << std::setw(5) << std::setprecision(3) << jacobian[i][j];
            }

          std::cout << std::endl;
        }

      // we can also calculate the jacobian of the reduced system
      // in a similar way
      pModel->calculateJacobianX(jacobian, 1e-12, 1.0);
      // this time generating the output is actually simpler because the rows
      // and columns are ordered in the same way as the independent variables of the state ↩
          temple
      std::cout << std::endl << std::endl << "Reduced Jacobian Matrix:" << std::endl << std ↩
          ::endl;
      std::cout << std::setw(5) << " ";
      CModelEntity* const* beginIndependent = stateTemplate.beginIndependent();
      const CModelEntity* const* endIndependent = stateTemplate.endIndependent();

      while (beginIndependent < endIndependent)
        {
          std::cout << std::setw(5) << (*beginIndependent)->getObjectName();
          ++beginIndependent;
        }

      std::cout << std::endl;
      beginIndependent = stateTemplate.beginIndependent();
      size_t iMax = stateTemplate.getNumIndependent();

      for (size_t i = 0; i < iMax; ++i)
        {
          std::cout << std::setw(5) << (*(beginIndependent + i))->getObjectName();

          for (size_t j = 0; j < iMax; ++j)
            {
              std::cout << std::setw(5) << std::setprecision(3) << jacobian[i][j];
            }

          std::cout << std::endl;
        }

    }

  return 0;
}
```

1.9.9 Calculating the jacobian of a model at the steady state

This example shows how to calculate the jacobian matrix of a model at the steady state state.

```
// Begin CVS Header
//   $Source: /fs/turing/cvs/copasi_dev/copasi/bindings/cpp_examples/example9/example9.cpp, ↩
    v $
//   $Revision: 1.2.2.3 $
//   $Name: Build-33 $
//   $Author: shoops $
//   $Date: 2011/03/30 16:00:39 $
// End CVS Header

// Copyright (C) 2011 - 2010 by Pedro Mendes, Virginia Tech Intellectual
// Properties, Inc., University of Heidelberg, and The University
// of Manchester.
// All rights reserved.

/**
 * This example is similar to example 8. We also calculate the jacobian,
 * but this time we want the jacobian at the steady state.
 * This is somewhat easier than calculating it manually in the model
 * becuase the steady state calculates it and we can get an annotated matrix which
 * tells us which coumn and which row represent what.
 *
 * So in this example, we learn how to work with annotated matrices.
 */
#include <iostream>
#include <iomanip>
#include <vector>

#define COPASI_MAIN
#include <copasi/copasi.h>
#include <copasi/report/CCopasiRootContainer.h>
#include <copasi/CopasiDataModel/CCopasiDataModel.h>
#include <copasi/model/CModel.h>
#include <copasi/model/CModelValue.h>
#include <copasi/utilities/CAnnotatedMatrix.h>
#include <copasi/steadystate/CSteadyStateTask.h>

const char* MODEL_STRING = \
                           "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n"
                           "<!-- Created by COPASI version 4.5.31 (Debug) on 2010-05-11  ↩
                               13:40 with libSBML version 4.1.0-b3. -->\n"
                           "<sbml xmlns=\"http://www.sbml.org/sbml/level2/version4\" level ↩
                               =\"2\" version=\"4\">\n"
                           "  <model metaid=\"COPASI1\" id=\"Model_1\" name=\"New Model\">\ ↩
                               n"
                           "    <listOfUnitDefinitions>\n"
                           "      <unitDefinition id=\"volume\" name=\"volume\">\n"
                           "        <listOfUnits>\n"
                           "          <unit kind=\"litre\" scale=\"-3\"/>\n"
                           "        </listOfUnits>\n"
                           "      </unitDefinition>\n"
                           "      <unitDefinition id=\"substance\" name=\"substance\">\n"
                           "        <listOfUnits>\n"
                           "          <unit kind=\"mole\" scale=\"-3\"/>\n"
                           "        </listOfUnits>\n"
                           "      </unitDefinition>\n"
                           "      <unitDefinition id=\"unit_0\">\n"
                           "        <listOfUnits>\n"
                           "          <unit kind=\"second\" exponent=\"-1\"/>\n"
```

```
"        </listOfUnits>\n"
"      </unitDefinition>\n"
"    </listOfUnitDefinitions>\n"
"    <listOfCompartments>\n"
"      <compartment id=\"compartment_1\" name=\"compartment\"  ↩
    size=\"1\"/>\n"
"    </listOfCompartments>\n"
"    <listOfSpecies>\n"
"      <species metaid=\"COPASI2\" id=\"species_1\" name=\"A\"  ↩
    compartment=\"compartment_1\" initialConcentration=\"1\"/>\n"
"      <species metaid=\"COPASI3\" id=\"species_2\" name=\"B\"  ↩
    compartment=\"compartment_1\" initialConcentration=\"0\"/>\n"
"      <species metaid=\"COPASI4\" id=\"species_3\" name=\"C\"  ↩
    compartment=\"compartment_1\" initialConcentration=\"0\"/>\n"
"    </listOfSpecies>\n"
"    <listOfReactions>\n"
"      <reaction metaid=\"COPASI5\" id=\"reaction_1\" name=\" ↩
    reaction_1\" reversible=\"false\">\n"
"        <listOfReactants>\n"
"          <speciesReference species=\"species_1\"/>\n"
"        </listOfReactants>\n"
"        <listOfProducts>\n"
"          <speciesReference species=\"species_2\"/>\n"
"        </listOfProducts>\n"
"        <kineticLaw>\n"
"          <math xmlns=\"http://www.w3.org/1998/Math/MathML\">\n ↩
    "
"            <apply>\n"
"              <times/>\n"
"              <ci> compartment_1 </ci>\n"
"              <ci> k1 </ci>\n"
"              <ci> species_1 </ci>\n"
"            </apply>\n"
"          </math>\n"
"          <listOfParameters>\n"
"            <parameter id=\"k1\" name=\"k1\" value=\"0.2\"  ↩
    units=\"unit_0\"/>\n"
"          </listOfParameters>\n"
"        </kineticLaw>\n"
"      </reaction>\n"
"      <reaction metaid=\"COPASI6\" id=\"reaction_2\" name=\" ↩
    reaction_2\" reversible=\"false\">\n"
"        <listOfReactants>\n"
"          <speciesReference species=\"species_2\"/>\n"
"        </listOfReactants>\n"
"        <listOfProducts>\n"
"          <speciesReference species=\"species_3\"/>\n"
"        </listOfProducts>\n"
"        <kineticLaw>\n"
"          <math xmlns=\"http://www.w3.org/1998/Math/MathML\">\n ↩
    "
"            <apply>\n"
"              <times/>\n"
"              <ci> compartment_1 </ci>\n"
"              <ci> k1 </ci>\n"
"              <ci> species_2 </ci>\n"
"            </apply>\n"
"          </math>\n"
"          <listOfParameters>\n"
"            <parameter id=\"k1\" name=\"k1\" value=\"0.1\"  ↩
    units=\"unit_0\"/>\n"
"          </listOfParameters>\n"
```

```
                              "        </kineticLaw>\n"
                              "      </reaction>\n"
                              "    </listOfReactions>\n"
                              "  </model>\n"
                              "</sbml>\n";


int main()
{
  // initialize the backend library
  // since we are not interested in the arguments
  // that are passed to main, we pass 0 and NULL to
  // init
  CCopasiRootContainer::init(0, NULL);
  assert(CCopasiRootContainer::getRoot() != NULL);
  // create a new datamodel
  CCopasiDataModel* pDataModel = CCopasiRootContainer::addDatamodel();
  assert(pDataModel != NULL);
  assert(CCopasiRootContainer::getDatamodelList()->size() == 1);
  // next we import a simple SBML model from a string

  // clear the message queue so that we only have error messages from the import in the  ↩
      queue
  CCopasiMessage::clearDeque();
  bool result = pDataModel->importSBMLFromString(MODEL_STRING);
  // check if the import was successful
  CCopasiMessage::Type mostSevere = CCopasiMessage::getHighestSeverity();
  // if it was a filtered error, we convert it to an unfiltered type
  // the filtered error messages have the same value as the unfiltered, but they
  // have the 7th bit set which basically adds 128 to the value
  mostSevere = (CCopasiMessage::Type)((int)mostSevere & 127);

  // we assume that the import succeeded if the return value is true and
  // the most severe error message is not an error or an exception
  if (result != true &&  mostSevere < CCopasiMessage::ERROR)
    {
      std::cerr << "Sorry. Model could not be imported." << std::endl;
      return 1;
    }

  // get the task list
  CCopasiVectorN< CCopasiTask > & TaskList = * pDataModel->getTaskList();

  // get the trajectory task object
  CSteadyStateTask* pTask = dynamic_cast<CSteadyStateTask*>(TaskList["Steady-State"]);

  // if there isn't one
  if (pTask == NULL)
    {
      // create a new one
      pTask = new CSteadyStateTask();
      // remove any existing steadystate task just to be sure since in
      // theory only the cast might have failed above
      TaskList.remove("Steady-State");

      // add the new task to the task list
      TaskList.add(pTask, true);
    }

  CCopasiMessage::clearDeque();

  try
```

```
  {
    // initialize the trajectory task
    // we want complete output (HEADER, BODY and FOOTER)
    pTask->initialize(CCopasiTask::OUTPUT_SE, pDataModel, NULL);
    // now we run the actual trajectory
    pTask->process(true);
  }
catch (...)
  {
    std::cerr << "Error. Running the scan failed." << std::endl;

    // check if there are additional error messages
    if (CCopasiMessage::size() > 0)
      {
        // print the messages in chronological order
        std::cerr << CCopasiMessage::getAllMessageText(true);
      }

    CCopasiRootContainer::destroy();
    return 1;
  }

// now we can get the result of the steady state calculation, e.g. the jacobian
// matrix of the model at the steady state
// here we can either get the jacobian as we did in example 8 as a matrix with
// getJacobian, or we can use getJacobianAnnotated to get an annotated matrix
// Corresponding methods for the reduced jacobian are getJacobianX and  ↩
   getJacobianXAnnotated
const CArrayAnnotation* pAJ = pTask->getJacobianAnnotated();
assert(pAJ != NULL);

if (pAJ != NULL)
  {
    // we do the output, but as in contrast to the jacobian in example 8,
    // we now have all the information for the output in one place

    // first the array annotation can tell us how many dimensions it has.
    // Since the matrix is a 2D array, it should have 2 dimensions
    assert(pAJ->dimensionality() == 2);
    // since the matrix has a dimensionality of 2, the index for the underlying abstract  ↩
       array
    // object is a vector with two size_t elements
    // First element is the index for the outer dimension and the second element is the  ↩
       index
    // for the inner dimension
    std::vector<size_t> index(2);
    // since the rows and columns have the same annotation for the jacobian, it doesn't  ↩
       matter
    // for which dimension we get the annotations
    const std::vector<std::string>& annotations = pAJ->getAnnotationsString(1);
    std::cout << "Jacobian Matrix: " << std::endl << std::endl;
    std::cout << std::setw(5) << " ";

    for (size_t i = 0; i < annotations.size(); ++i)
      {
        std::cout << std::setw(5) << annotations[i];
      }

    std::cout << std::endl;

    for (size_t i = 0; i < annotations.size(); ++i)
      {
```

```
          std::cout << std::setw(5) << annotations[i];
          index[0] = i;

          for (size_t j = 0; j < annotations.size(); ++j)
            {
              index[1] = j;
              std::cout << std::setw(5) << std::setprecision(3) << (*pAJ->array())[index];
            }

          std::cout << std::endl;
        }


    }

  CCopasiRootContainer::destroy();
  return 0;
}
```