MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Writing Graph Algorithms in Executable Pseudocode

BACHELOR THESIS

Boris Ranto

Brno, spring 2011

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Boris Ranto

Advisor: Ing. Petr Gotthard

Acknowledgement

I would like to thank Vojtěch Řehák for his testing and corrections that he suggested.

Abstract

The aim of this bachelor thesis is to provide a tool that would allow software designer to write pseudocode that is directly executable without additional complexity in terms of pseudocode simplicity. The thesis is aimed to serve as a learning material for software designers that are willing to comply to the suggested notation.

Keywords

scstudio, graph algorithms, python, pseudocode, BMsc, HMsc, checking algorithms

Contents

1 Introduction

The main center of interest of this thesis is project scstudio (The Sequence Chart Studio)[1, 2]. Scstudio is a tool for drawing sequence charts and verifying properties on them. This thesis focuses more on the second part - verification algorithms. The final aim is to be able to write high-level executable code for verification of properties on message sequence charts (MSC).

The thesis is divided into four basic chapters. First chapter specifies the notation for writing pseudocode. The chapter serves as an introduction to the high level language that will be used. The notation is based on Python[3, 4] programming language and tries to extend it so that pseudocode close to the mathematical notation can be written. The notation in this chapter should describe all the parts of the language that are necessary for the purposes of this thesis along with probably most important part - sets.

Second chapter describes the way Python's data model had to be extended. The data model is extended so that objects from scstudio can be created and modified. At the end of the chapter modification of built-in set data type is described. The original built-in set can't handle highly dynamic objects for complexity reasons and therefore more high-level implementation of sets had to be provided. Since Python is highly object oriented programming language by extending the data model the language itself is extended too. Final part of this chapter describes creation of wrapper for original sets and newly defined sets.

Third chapter describes the implementation of checking algorithms in executable pseudocode. The chapter contains a lot of examples so that reader can see how to write the code. The description of implementation of all the checking algorithms (except for race checker) that were defined in the Master's thesis of Jindřich Babica[5] is provided in this chapter.

Fourth chapter of this thesis is dedicated to the integration with existing scstudio project. The aim was to simplify the integration as much as possible so that virtually no C++ code had to be written. This integration simplifies testing and input handling. The integration has two parts. The first one is developer level integration that provides higher level of configurability for checkers. The second one is user level integration that is aimed for end users of scstudio. This means that it does not expect user to do any compilation of scstudio or any additional modules.

I hope that this thesis will help a lot of software designers ease up their work and it will remove the necessity for implementation of checking algorithms in low level C++ code.

The result of this thesis is the extension of the Python data model, extension of scstudio and the bachelor thesis itself that will hopefully serve as a good learning material. The extension is divided into several modules although there is also module that can import the complete extension called pysc.

The resulting extension is licensed by *GNU/LGPL* 3.0[6]. Therefore the extension is free software and can be legally modified to suit the needs of any other project with respect to the license.

2 Notation for writing high-level Msc algorithms

There is quite a lot of languages that are high-level and can handle a lot of operations that are necessary for writing high-level graph algorithms. Therefore it is quite wise to pick one of the languages and build entire notation upon the language. I've decided to pick Python programming language for this purpose and therefore syntax for the writing of high-level code is mostly determined by Python's syntax. It must comply to the same basic rules as Python programs must although it might differ in some cases. It might be more strict than original Python's syntax and in few areas even less feature rich. Please take this notation as a recommendation for writing graph algorithms for the project scstudio. If you're a skilled python programmer you might want to use techniques that are not described in this thesis although you must be aware of the fact that person that learned only this notation might not understand your code. This chapter is aimed primarily for people that are not familiar with Python Programming language (with an exception in the section that describes set operations, the section describes primarily new features).

Python was chosen as a basis because the language was already created with high-levelness in mind. It also supports (at least to certain point) all three major programming paradigms (functional programming, standard structured programming and object oriented programming). Although the object oriented paradigm is not well suited for writing graph algorithms in pseudocode (and it is not part of the notation) it comes in very handy if the aim is to create high-level extension of the language. Python is also known for its readability.

More specifically version 3 of Python programming language was chosen as a basis. Although this version is mostly backward compatible there are certain incompatibilities (most notably: function print can't be called without parenthesis and indentation must be more consistent (without mixing tabulators and spaces)). The code and library itself is written so that it can be executed in both environments with few minor inconsistencies in the output of algorithms (function print in Python 2 might consider its input enclosed in parenthesis as a tuple). Another incosistency between the two Python versions is that Python version 3 started using unicode as default encoding for strings. Therefore if Python 2 is used it is the responsibility of an user to use unicode strings for labels.

Variable names in examples (where it is useful) are often shortcuts from their full names and therefore should be quite self-explanatory.

2.1 Indentation

The language does not use any begin/end syntax that would specify the begin and the end of the block of code. The code block is defined purely by indentation and therefore it is necessary to define a way to indent algorithms. That is why this notation forces indentation of the code blocks. Tabulators are preferred over spaces. The indentation steps and whole indentation must be consistent through whole document.

2.2 Basics

This section is just a very slight introduction to the notation and should give the reader just a basic feeling about the way algorithms are written. This part is here to avoid at least some confusion with the problematic cyclic dependencies of definitions.

All the values are assigned with =. When already existing variable is assigned a value, its previous content is unreferenced (and can be deleted from memory).

Everything is considered to be an object. There are two basic kinds of objects: function objects and ordinary objects. Function objects can be called using parenthesis. Any function object can be assigned a value of any other (function) object.

To print any value, function 'print' can be used.

Natural and rational numbers are defined. Strings are also defined and are denoted by quotation marks.

Line beginning with >>> in examples denotes an expression. Output of the expression is in the next line. Comments begin with # sign.

Example

```
>>> a = 1
>>> str = "help"
>>> print(str, a) # Print the variables separated by
    space
help 1
```

2.3 Module import

Every source file can also be treated as a module. There are three ways to import any module (source file) although one of them is just an extension of another one.

import pysc

This call will create object named pysc (name can be changed) that will contain everything the pysc module contains. Therefore any part can be accessed directly through dot operator.

Example: If one wants to access attribute alpha from module pysc, it is necessary to precede the attribute with `pysc.` and the resulting statement will look like this: `pysc.alpha`.

from pysc import *

This will import everything from the pysc module directly into program. Therefore to access any object, all that is needed is to use its name.

Example: To access attribute alpha, it is necessary to use statement 'alpha'.

import module as ownName

This call will just rename the module name to ownName, everything else is same as in first case.

2.4 Basic binary operators

Several predefined operators exist. The operators can be recognized by their infix notation. The most basic non-conditional operators include '=' for assignment, + for addition, * for multiplication, - for subtraction, / for division and ** for exponentiation.

Example

```
>>> n=7
>>> print(2**n, 2.0**7, n)
128 128.0 7
>>> a = n+3
>>> print(a*n, a, n)
7 1 7
```

2.5 Expressions

Expression is any object or any usage of any object (mostly function objects) or usage of operators on objects. Expression's objects must contain necessary attributes. Everything after »> in previous examples was also an example of expression (although the line can be a little more powerful as described in next section).

2.6 Body

Body is basic element of any program. It defines complete program. Basically body denotes any expressions (+ special expressions) that use at least the indentation of the first expression. Since it is defined recursively, any significant part of body is also a body (and working program).

Every body must satisfy any of following scheme (recursively):

```
——————————
exp1
——————————
spExp:
    body
——————————
body1
...
bodyK
——————————
```

Body in any of the schemes can denote any other scheme. spExp denotes condition, loop, cycle or function definition. These special expressions can be recognized by following keywords: if, for, while, def (as defined later in this chapter).

2.7 Objects

Standard objects

Everything is considered to be an object having some properties (that are also objects). Properties of any object can be accessed through dot operator. Properties of any object can be viewed with dir() function object. Please note that every property of object is also a object and therefore can be treated same way as the new object.

Every property call must satisfy following scheme:

```
object.property
```

Function objects

Special types of objects are so-called function objects. Function object is just like any other object except for the fact that it can be executed using parenthesis.

Function objects are commonly referred to as functions.

Functions are defined using 'def' keyword. Functions can return a value using special expression 'return Exp1' where Exp1 is any regular expression.

Every function declaration must satisfy following scheme:

```
def function(par1, ..., parN, oPar1=v1, ..., oParM=vM)
  :
    body
```

parX parameters are mandatory, oParX parameters are voluntary. If any voluntary parameter is not specified in function call, default value vX is used for the parameter.

Every function execution must satisfy one of the following schemes or their mixture:

```
f(v1, ..., vN, ov1, ..., ovK, pX=opv1, ..., pM=opvS)
```

Character f denotes function that is called, vX denotes value for X-th mandatory parameter, ovX's denote values for first K voluntary parameters. Any voluntary parameter can be directly specified with pX=opvX notation.

2.8 Conditions

Conditional operators

These operators are usually used in conditions because they map their input into boolean value. They also use infix notation (as well as any other operator). Conditional operators include == for test on equality, != for test on non-equality, >, <, <=, >= for comparing, 'in' and 'not in' for test of membership to a complex object and 'is', 'is not' for the test of identity of two objects.

Conditional operators can also be from boolean to boolean (these are usually used to merge several conditions). This group includes 'and' and 'or' operators.

Condition expressions

Any object can be condition expression. Object is considered to be true unless it is not empty (False, 0, empty list, empty string, empty object=None, ...). Every condition expression must satisfy any of the following schemes:

```
object
————
condExp binOp condExp
————
not condExp
```

binOp denotes binary operation, condExp denotes any of the preceding.

Complete conditions

Condition must satisfy following scheme:

```
if condExp:
    body1
else:
    body2
```

Else part is voluntary. If condExp is satisfied, body1 is executed, otherwise body2 is executed. Please note that it is possible to use shortcut notation of elif for 'else if' branch. This notation allows to save additional indentation.

2.9 Loops and cycles

Every loop must satisfy following scheme:

```
for member in iterator:
    body
```

Iterator denotes object that can be iterated (most notably lists and sets). Object member will be accessible in the body of this expression. Both member and iterator can be modified in the body.

Cycles are defined basically the same way as in most common imperative programming languages.

```
while condExp:
    body
```

If conditional expression condExp is satisfied body is cyclically executed until condExp is not satisfied.

2.10 Lists

List is very powerful built-in type of objects. Lists can be recognized by the type of parenthesis that wrap them. Every list is ordered. The order is defined by the way the members of list are included into the list (usually appended). Lists can be iterated and are iterated with respect to the order in which they were defined.

Every list must satisfy following scheme (although it might be defined in few different ways it always satisfies the structure internally):

```
[object1 ,... ,objectN]
```

Members of list don't have to be of an identical type.

List indexation

Since lists are ordered they can be indexed. Lists are indexed by integers. Negative numbers are interpreted as indexation from the end of the list. First member is indexed by 0, second by 1, last by -1 and so on. Lists can also be indexed by intervals using : operator. Please note that left integer is inclusive and right integer is exclusive.

Function len(L) can be used to get the length of the list L.

List as a stack

To operate with a list as with a stack there is predefined function attribute 'append' to insert item to the end of the list and function attribute 'pop' to return and remove last item from the list.

Example

```
>>> a = [1,3,2,7]
>>> a.append('g')
>>> print(a[3], a[1:-1], a.pop())
7 [3, 2, 7] g
```

2.11 Tuples

Tuple can be thought of as a fixed list meaning that the size of tuple cannot be changed once the tuple is created. The benefit for using tuples is mostly in their speed with respect to lists.

Example

```
>>> (a, b, c) = (3, 4, [1, 2, 3]) # assigning values
    to three new variables
>>> print(c, a)
[1, 2, 3] 3
```

2.12 Sets

Sets can be created using optional initialization list and Set() function (without the list empty set is returned). Therefore to make set out of a list it is necessary to use wrapper function Set() that will remove duplicities and return particular Set object. The call must satisfy one of the following schemes:

```
a = Set([obj1, ..., objN])
b = Set()
```

There is no limitation on obj1, ..., objN although complexity of set operations might differ based on the type of these objects.

Sets are outputted with regular curly brackets that wrap them.

This implementation of sets is part of the module that is supposed to extend Python programming language so that the extended language can be as high-level as possible. Implementation of this set can be found in file list_ops.py.

Conditional operators over sets

Sets support all the basic operators for testing membership (in, not in) and comparing sets (`<, <=, >, >=, ==, !=`) meaning the relations of any `a` being subset of any `b` (`a <= b`) and similar.

Basic set operations

All the basic set operations are supported. Result of every operation is new set. No input set is modified when new set is being created.

Following schemes must be used to make union, intersection, set difference or symmetric difference:

```
set3 = set1 | set2 # Union
set4 = set1 & set2 # Intersection
set5 = set1 - set2 # Set difference
set6 = set1 ^ set2 # Symmetric difference
```

There is also 's + m' operator construction that adds new member m to the set s and returns new composed set. Its semantic is equivalent to `s | Set([m])`.

Transitive closure

A little more advanced operation of transitive closure of any set containing tuples (the standard set relation) is defined as the attribute of the Set. The function returns new set that is transitive closure of the original set. The attribute that is used for this is `transitive()`. It does not take any argument but it is rather complex attribute and therefore it is defined as a function attribute.

Set modification functions

The content of a set can easily be modified using few basic functions for addition and removal of set members.

For insertion of a member `m` to set `S` there is function attribute insert. Alternatively += operator is defined to simplify this process.

To remove a member of set there is the function attribute remove.

Example

```
>>> a = Set([1,3,5,7,8])
>>> a += 10
>>> a.remove(10)
>>> b = Set([2,3,5,12,15,2])
>>> print(a & b)
{3, 5}
>>> union = a | b
>>> print(union, union-b-a, union-a)
{1, 2, 3, 5, 7, 8, 12, 15} {} {2, 12, 15}
>>> sym_diff = a ^ b
>>> print(sym_diff, union & a)
{8, 1, 7} {8, 1, 3, 5, 7}
>>> print(union & a == a, a < union, a < b, a == a, a
    > b, union >= b)
True True False True False True
>>> print(b+14)
{2, 3, 5, 12, 14, 15}
>>> print((Set([1,2]) * Set([1, 2]) + (2,3)).
    transitive())
{(1, 2), (1, 3), (2, 1), (2, 3), (2, 2), (1, 1)}
```

2.13 Functional programming tools

Since the notation should be close to standard mathematical notation few functional programming concepts are introduced to allow closer level of expressibility.

Generating lists

List (or tuple) can be generated by a list with a condition. In order to create list (or tuple or set) A containing results of expression E that might depend on a member X of a iterable object L while satisfying condition C, following list generator can be used:

```
A = [E for X in L if C] # List A
A = (E for X in L if C) # Tuple A
A = Set([E for X in L if C]) # Set A
```

Standard operators counterparts

Every operator has its counterpart defined in operator module. This counterpart is defined just like a regular function and can therefore be passed as a parameter to a function. This is quite essential for functional programming.

Standard operators `<, <=, >, >=, is, is not, ==, !=` have following function counterparts defined in module operator: `lt, le, gt, ge, is_, is_not` respectively.

Full list of the operators counterparts is available online[7], for this thesis only counterparts that are mentioned here are necessary.

Standard functions

There are several functions quite essential for functional programming. One of the them is function map(A, B) that map function A on every member of iterable (list, set...) B.

Another very important function is function filter(F, I) that filters iterable object I for the members that satisfy condition F(X) where X is a member of I.

Function range(I, J, K) is defined and it returns a list that contains numbers from I to J-1 that are K steps away from each other. If K is not specified it is considered to be 1. If I is not specified (function range got only one attribute - J) it is assumed to be 0 resulting in the list that is from 0 to J-1.

3 Extending data model

3.1 Introduction to Msc

The MSC is a shortcut for Message Sequence Chart. It was defined by ITU-T[8] in Z.120[9] standard (recommendation). It can be described in the following way:

> MSC is a graphical and textual language for the description and specification of the interactions between system components. The main area of application for Message Sequence Charts is as an overview specification of the communication behaviour of real-time systems, in particular telecommunication switching systems. Message Sequence Charts may be used for requirement specification, simulation and validation, test-case specification and documentation of real-time systems.[10]

MSC consists of two languages: Basic Message Sequence Chart (BMSC) and High level Message Sequence Chart (HMSC). BMSC defines a sequence of messages that are passed between the systems while HMSC defines the relations between BMSCs.

MSC can be represented in two standard ways: by its textual form and by its graphical form. In this chapter the textual form of MSC is described. The graphical form is expressed by Microsoft Visio[11] add-on.

Python's data model had to be extended so that all the Msc objects can be defined in the language. All the attributes that take no argument are not function attributes and therefore these attributes can be get directly without the parenthesis. These attributes (if it makes sense to set the attribute - attribute is not computed from other attributes) can also be set with standard notation of `object.attribute = value`.

The knowledge of Master's thesis of Jindřich Babica[5] is necessary for complete understanding of this section.

3.2 BMsc

The object `BMsc` creates and manipulates BMsc. The call of `BMsc()` can be used for the creation of the BMsc. The object consists of several attributes. Following attributes are defined for the object:

- `type` being the attribute that contains information about the type of Msc, for BMsc it contains the string BMsc

- `label` representing the label of the Msc
- `instances` being the set of the `Instances` for the `BMsc`
- `instance` where the instance that will be assigned to this attribute will be added to the set of instances
- `events` that is the attribute that is supposed to be the equivalent of $Event_b$, the attribute returns the set of all the events referenced in the BMsc
- `CoregionEventRelation` being the attribute that returns the equivalent of $CoregionEventRelation_b$ (set of CoregionEventRelations in the `BMsc`)
- `<Type>` to test whether Msc is of given type, possible values of `<Type>` are `BMsc` and `HMsc`

To completely fill the object BMsc only attributes label and instances (by direct manipulation or with the use of the instance attribute) has to be specified.

BMsc and its methods are implemented in the file msc.py.

3.3 Instance

BMsc is specified by process instances. These instances are represented by the object Instance. This object can be created by the call `Instance()`. Instance as an object contains following attributes:

- `areas` for the set of its areas
- `area` to add an area to the instance
- `form` containing the form as a string, possible values are Line and Column
- `label` for the label of the instance
- `kind` for the kind/type of the instance
- `type` containing the same value as the `kind`
- `first` for the first EventArea
- `last` for the last EventArea

- `bmsc` for the BMsc that this instance is part of
- `line_begin` for the tuple of two numbers representing the beginning of line
- `line_end` for the tuple of two numbers representing the end of line
- `width` for the width of the instance
- `height` for the height of the line, computed from `line_begin` and `line_end`
- `is_empty` to test whether the `last` event is defined
- `has_events` to test whether there is any event

Additionally function attribute `add_area(a1)` is defined that inserts area `a1` to the instance.

To completely fill the instance the attributes areas, form, label, line_begin, line_end, width and type/kind (just one of them) need to be specified. The attribute `areas` shall be filled by the attribute `area` or the function attribute `add_area`.

Instance is implemented in file instance.py.

3.4 Areas

The Instances are specified by EventAreas - data types representing areas of the events. Two basic types are defined: CoregionArea, StrictOrderArea.

Their definition is divided into two parts: common part and specific part. Therefore few attributes are common for both areas:

- `next` for the next area, this attribute can also be set
- `previous` for the previous area, this attribute can also be set
- `begin_height` for the integer specifying the height at the beginning
- `end_height` for the integer specifying the height at the end
- `width` for the width of the area
- `instance` for the instance that the area belongs to
- `is_empty` for the test whether the area contains anything

- `is_first` for the test whether the area is first
- `height` for the height (computed as the absolute value of the difference of the begin_height and the end_height)
- `events` for all the events in the corresponding area
- `type` returning the string representing the type of the area
- `<Type>Area` to test the type of the area, possible values of the `<Type>` are StrictOrder and Coregion

Function add_event is also defined for both types of areas although it is overloaded and a little more sophisticated for CoregionArea (where it can add the event to both the minimal and the maximal events).

Only the attributes width, begin_height, end_height and next/previous need to be set.

The StrictOrderArea does not extend the common attributes too much. Only two additional attributes are defined:

- `first` for first event in the area
- `last` for last event in the area

Both attributes can and also should be set (either directly or indirectly using the function add_event).

The CoregionArea is a little more complex, it has following attributes defined:

- `form` for the form of the drawn coregion
- `minimal_events` for the set of its minimal events
- `minimal_event` to add new minimal event
- `maximal_events` for the set of its maximal events
- `maximal_event` to add new maximal event
- `events` returning union of minimal and maximal events

Basic functions for manipulating minimal_events and maximal_events sets are also defined although these sets can be manipulated directly with standard set functions or operators. The functions are quite self-explanatory: add_minimal_event, remove_minimal_event, add_maximal_event and remove_maximal_event.

Only following three attributes can and also should be set (either directly or using functions/attributes mentioned above): form, minimal_events, maximal_events.

EventAreas are implemented in file area.py.

3.5 Events

EventAreas consist of Events. For each EventArea's type there is corresponding Event's type. These basic types of events are known as CoregionEvents and StrictEvents.

Both types of events have quite a lot of common attributes:

- `position` being the relative position of the event to the respective instance/area
- `message` referencing the message corresponding to the event, it might be Complete or Incomplete message
- `area` that holds the reference to the area that the event belongs to
- `complete_message` returning the message if it is CompleteMessage
- `incomplete_message` returning the message if it is IncompleteMessage
- `matching_event` returning matching event if the message is CompleteMessage (`None` is returned otherwise)
- `is_matched` testing whether the message is CompleteMessage
- `is_send` checking whether the event is the sender or the message was lost
- `is_receive` being the negation of the `is_send` if a message exists
- `receiver_label` returning the label of the receiver if it exists and the instance label of the message or instance label of the event otherwise
- `sender_label` returning the sender label if possible (falling back to the instance label of the message or the instance label of the event)
- `general_area` that is just an equivalent of the area
- `instance` returning the instance of the area of the event

- `<Type>Event` to test the type of the event, the `<Type>` can be either Strict or Coregion

Only the attributes position and message need to be set for both types of the events.

StrictEvent contains all of the attributes above and adds several new attributes that are specific for the StrictEvent:

- `successor` expressing the following event
- `predecessor` referencing the previous event
- `is_first` testing whether the event is first (have no predecessor) in its area
- `is_last` checking whether the event is last (have no successor) in its area

The attribute successor can and should be set (along with the common attributes) to completely fill the StrictEvent (predecessor attribute is handled automatically when the successor is being set).

As well as StrictEvent, CoregionEvent has several additional attributes that are specific for it:

- `successors` meaning the set of successors (CoregionEventRelations) of the event
- `successor` to add new successor by setting this attribute to any new successor, predecessors are handled as well
- `predecessors` referencing the set of the predecessors of the event
- `is_minimal` returning true if there are no predecessors
- `is_maximal` returning true if there are no successors
- `has_successors` testing for the presence of any successors
- `has_predecessors` checking whether there are any predecessors
- `coregion_area` being the equivalent of the `area`

Additionally functions add_successor(succ) and remove_successor(succ) are defined for CoregionEvent to add and remove any successor.

Only the sets successors and predecessors need to be filled (along with the common attributes) to completely fill the CoregionEvent. These attributes can be filled with the use of the successor modifying functions (or attribute) mentioned above.

CoregionEventRelation is simply defined as an object containing three attributes: successor, predecessor and line. Successor and predecessor are CoregionEvents and line is just a list of tuples containing coordinates x and y, for example `[(0, 0), (1.1, 1.3)]`. The CoregionEventRelation is filled when all the three attributes are set.

The events are implemented in file event.py.

3.6 Messages

Events carry messages. These messages are represented by two objects depending on the type of the message. The two types of the messages are CompleteMessage and IncompleteMessage.

Each of the message types has its own attributes defined. CompleteMessage has following attributes defined:

- `otype` being the attribute that stores the string that defines the type of the object (specifically message), for CompleteMessage its value is "Complete", for IncompleteMessage it's "Incomplete"
- `label` being the string that labels the message
- `send_event` referencing the event that sends the message
- `receive_event` referencing the event that receives the message
- `sender` being the instance of send_event
- `receiver` being the instance of receive_event
- `is_glued` to test whether the events are already set
- `events` being the attribute that can set both send event and receive event, it is assigned a tuple of the template `(send_event, receive_event)`
- `<Type>Message` to test whether the message is of the given type, possible values of the `<Type>` are Complete and Incomplete

Only the attributes label, send_event and receive_event (either directly or using the attribute events) need to be set.

IncompleteMessage contains attributes type, <Type>Message, is_glued and label that have the same meaning as in the CompleteMessage case. There are also attributes specific for this type of the message:

- `type` being the attribute of the type string that stores the type of message - whether the message is lost or found
- `dot_position` denoting the coordinates of the message
- `instance_label` expressing the label of its instance
- `event` referencing the event connected to the message, the attribute can be set directly and the message will be appropriately glued
- `is_lost` to test whether the message was lost
- `is_found` to test whether the message was found

Only the attributes label, type, event, dot_position and instance_label need to be set to completely fill the IncompleteMessage.

Both CompleteMessage and IncompleteMessage are implemented in file messages.py.

3.7 HMsc

The object `HMsc` creates and manipulates HMsc. For creation of the HMsc the call `HMsc()` can be used. `HMsc` consists of several attributes. Following attributes are defined for `HMsc`:

- `type` being the attribute that contains the information about the type of Msc, for HMsc it contains the string hmsc
- `nodes` being the set representing all the nodes that are in HMsc (except for StartNode)
- `start` that holds the reference to the StartNode of the HMsc
- `snodes` being the set containing all the nodes (including the start node)
- `label` representing the label of the Msc
- `node` being the attribute for the addition of new node to `Msc`, by assigning a value to the attribute new node will be added to the nodes attribute
- `<Type>` to test whether BMsc is of given type, possible values of the `<Type>` are `BMsc` and `HMsc`

To completely fill the HMsc only attributes nodes (by manipulating the set or with use of the attribute node), start and label need to be specified.

HMsc and its methods are implemented in the file msc.py.

3.8 Nodes

Every HMsc contains a list of nodes. The nodes in HMsc usually represent other Mscs. In order to create and manipulate nodes, the object named Node is defined.

Its first parameter is string declaring the type of the node. Possible values are `"Reference"`, `"Start"`, `"End"`, `"Condition"` and `"Connection"`. If the parameter is not passed, the Node is considered to be StartNode.

As an alternative, all the nodes can be created with their full names, for example `ReferenceNode`. This form is just a shortcut for the previous form.

Examples of usage of the data model can be found in the chapter that deals with implementing checking algorithms.

Every Node holds following basic attributes: owner, type, succ, pred and msc. These attributes can be accessed and set directly. Please note that not all of the attributes make sense for every type of Node.

- `owner` that holds the reference to the owner of the node (the HMsc that the node is part of)
- `type` that is the string referencing the type of the Node, value is same as the string that the node was created with
- `pred` that is the set of predecessors of current node, it holds that A is predecessor of B if and only if B is successor of A
- `succ` that is the set of successors of current node according to node relations
- `successor` that can add new successor to the node
- `msc` that is the reference to another Msc, this attribute is only used if the type of the node is ReferenceNode
- `<Type>Node` that is the attribute returning boolean value (True if and only if the node is of the type <Type>), <Type> can be any of the Reference, Start, End, Condition and Connection

- `reachable` returning the list of reachable nodes from given node with respect to the path notion as defined in Deadlock checker part of [5]
- `label` for the label of the node
- `position` for the tuple representing position of the node

Attributes owner, type, msc, label and position can (and should) also be set. Attributes succ and pred can be modified using set operations or by the successor attribute. All the other attributes are only computed (and cannot be set).

Attributes `succ` and `pred` can be used for traversing. Some basic traversers can be defined for nodes. Simplest one is probably `reachable`. It is node function that returns list of all the nodes where there is a path from the present node to the node that is member of the returned list. This traverser is defined as an attribute of Node. Traversers creation process will be better explained in the chapter that describes implementation of checking algorithms.

Additionally functions `has_pred(a)` and `has_succ(b)` are defined to test whether object has predecessor `a` or successor `b`. These functions are just shortcuts for `a in c.pred` and `b in c.succ`.

The described objects are implemented in the file node.py.

3.9 Channel mapping

Channel mapping serves as a tool that defines the way messages are ordered through FIFO (First In First Out) channels. Two basic ways to define message order are declared: `SRChannelMapper` and `SRMChannelMapper`. The SRChannelMapper has one FIFO channel per communicating pair while the SRMChannelMapper has one FIFO channel per communicating pair and message name.

Two basic data types are important for this section: `MessageParts` and `ChannelMappers`. Both of these are just virtual data types with real implementations in `SRMessagePart`, `SRMMessagePart`, `SRChannelMapper`, `SRMChannelMapper`.

Both implementations of MessageParts need one attribute when they are created. The attribute is event.

`SRMessagePart` is the simpler `MessagePart` since it contains only two standard attributes, one function attribute and one operator redefinition. The standard attributes are sender for label of sender and receiver for label of receiver. The function attribute is `same_channel(e1, e2)` and it tests whether the two events are on same channel. Operator < is redefined so that MessageParts can be compared.

`SRMMessagePart` contains everything that `SRMessagePart` does and one additional standard attribute: `label` for label (name) of the message of the event. Operator < and function attribute `same_channel` are redefined to correspond to the modified standard attributes.

`SRChannelMapper` and `SRMChannelMapper` are defined almost the same way. Both of them contain function attributes `channel(event)` and `same_channel(e1, e2)`. The attribute `same_channel` is same as in the `MessagePart` case (respectively for the corresponding ChannelMappers). The attribute `channel` returns the index of event in channel if the event is in the channel. If the event is not found, it is added and corresponding index is returned (length of the list that contains channels).

3.10 Set implementation

The language itself provides definition of sets. The problem is that the object is incomplete because it can handle only certain kind of input (because of the way the set is implemented). The set uses efficient implementation that is based on comparison of its object with respect to their hashes. The problem is that several data types can't have their hashes defined because of their very dynamic nature. The typical representatives of objects that are not hashable are lists and modifiable sets. This division would bring unwanted complexity to the notation and therefore sets that would be closer to the mathematical sets had to be defined.

Python allows a person to modify itself very easily. Mostly thanks to its cleanness and operator/function overloading. Python considers operators to be just a syntactic sugar for function attributes of objects and therefore by overriding attributes, operators can be overridden, too.

Inefficient set implementation

Python supports basic set objects but these objects are not complete and have their limitations that make work with sets a little low-level. Because of complexity issues (there must be some sort of comparison to create efficient sets) every member of built-in set object must be hashable. A nice example of object that is not hashable is `list` or `set` itself (although there is object `frozenset` that is hashable). Basically non-hashable objects are dynamic objects that have redefined equality operator (otherwise pointer-based hash can be used).

Since this issue makes usage of sets quite nonintuitive and low-level, new implementation of sets had to be introduced. New sets that can work without comparison had to be implemented. Additionally wrapper class for complete set and built-in set had to be created. The wrapper class uses efficient sets whenever it can and fall back to inefficient no-comparison based sets otherwise. Therefore the set can work for every object. The inefficient set is called `nefset`.

The new object `nefset` is derived from standard `list` object and therefore all the operations that are not overridden can still be used (for example attribute `remove`).

All the common set operations should be defined. Following operations and functions were overridden or defined so that intuitive version of sets can be defined.

- `__repr__` is function that tells python how the object should look when it is outputted, function is defined so that it can remind more known representation of sets, therefore members are enclosed by curly brackets
- `__eq__` that redefines equality operator on sets, sets are equal if every member of set a is in set b and vice versa
- `__ne__` that redefines `!=` operator, test for non equality of two sets
- `__le__` that redefines <= operator to the mathematical operation of $\subseteq$
- `__lt__` that redefines < operator to the mathematical operation of $\subset$
- `__ge__` that redefines >= operator to the mathematical operation of $\supseteq$
- `__gt__` that redefines > operator to the mathematical operation of $\supset$
- `__or__` that maps operator `|` to the set operation of union: $\cup$
- `__and__` that maps operator `&` to the set operation of intersection: $\cap$
- `__sub__` that maps operator `-` to the set operation of difference: $\setminus$
- `__xor__` that maps operator `^` to the set operation of symmetric difference

- `__mul__` that maps operator `*` to the cartesian product of two sets
- `insert` that is the only attribute that is called directly and needed to be redefined since set can insert a member only if the member is not already there, the new member is the only attribute of this function

Wrapper for sets

As was mentioned earlier, the goal is to define sets that can be used for every object that can be created in Python and still be efficient if the members allows us to do so. Therefore it is necessary to create new wrapper class for built-in `set` object and previously defined `nefset` object.

Since `nefset` object was defined so that it uses same operators for same mathematical operations as `set` does, the wrapper class can be quite straightforward (in the cases when there is no mixing of internal types).

All the operations supported by this Set are mentioned in the chapter that specifies whole notation for writing graph algorithms (except data model for Msc objects).

The wrapper class holds only two attributes that specifies it. First of them is `hashable` and this attribute tells what kind of underlying set implementation is used. If the attribute `hashable` is `True` then function uses `set` otherwise `nefset` is used. Second attribute is data and it stores the created `set` or `nefset`.

Newly defined `Set` tries to use built-in `set` whenever it is possible and use `nefset` otherwise. Furthermore result of any of operations over `Set` uses `set` whenever it is possible and beneficial (for example, the result of intersection of internal `nefset` and internal `set` is always `set` since at least one of the attributes is `set` internally).

Most of the implementations of binary methods differentiate three basic states. First state is when attribute hashable is same for both objects. Standard operations defined on `set` or `nefset` can be used in this case. Second state is when first object is hashable and second is not. Third state deals with the case when first object is not hashable but second object is. This two states are dealt appropriately so that `set` is used internally if it is possible and beneficial.

Since mixing of internal representation is usually a special case, it might bring additional complexity.

Sets are implemented in file list_ops.py.

4 Implementing high-level algorithms

All the definitions in this part are referenced from the master's thesis of Jindřich Babica[5] and are denoted to the checking algorithms. The referenced checking algorithms are defined in this chapter with an explanation of their definition in suggested notation. This part should show the reader how to write code the way it was defined in previous chapters. Most of the pseudocode in this part is here to serve as an example of the usage of extended data model and the notation itself.

4.1 Basic transformation rules

The original algorithms are written in the mathematical notion. Therefore most important object is set object. The algorithms use sets in two ways. The first way references regular sets and this can be handled directly by the object `Set`. The other way is to test for the type of the object (the mathematical notion uses abstract sets that contain every member that is of the type, for example HMsc that contains all the HMscs). This can be handled by appropriate attribute (every object has got the appropriate boolean attributes defined).

The sets can also be iterated. The mathematical notion uses two standard iterators: $\forall$ and $\exists$. Both of these iterators can be handled by the standard for loop: `for member in set`.

The standard symbol $\in$ is used in two situations. To test whether $x \in set$ or to iterate over the set using for loop. Both of these situations can be handled directly by the $\widehat{\text{in}}$ statement.

The implication can be handled using `if` statement or using simple logic ($a \Rightarrow b \Leftrightarrow \neg a \vee b$) depending on the context and suitability.

All the standard operators ($\vee$, $\wedge$, $\neg$, $\cap$, $\cup$, ...) are handled by the corresponding operators as defined in the chapter that defines the notation (`or`, `and`, `not`, `&`, `|`,...).

New algorithms are written in the form of the functions taking necessary arguments. Every definition can be transformed into the function (with less or more modifications).

4.2 Deadlock checker

Definition 4.1 from thesis of Jindřich Babica defines subset of all the hmsc's. It's defined recursively and its basis is the input itself. Therefore the following function can be defined:

```
def referenced(h):
    if h.HMsc:
        # Initialization, step 1, h is in referenced(h
          )
        ref = [h]
    # Recursive part, over all the HMsc's that are (or
        will be) part of the referenced(h) and all its
        nodes, we're looking for their msc's and
      adding new HMsc's
    for k in ref:
        for n in k.nodes:
            if n.ReferenceNode and n.msc:
                msc = n.msc
                if msc.HMsc and msc not in ref:
                    ref.append(msc)
    return ref
```

Definition 4.2 defines simple property of being BMsc-graph (being HMsc and being the only member of the own referenced set). Following function can directly be defined:

```
def isBMscGraph(h):
    if h and h.HMsc and referenced(h) == [h]:
        return True
    else:
        return False
```

Definition 4.3 defines path. Since every node knows its successors and its predecessors it's quite easy to traverse nodes. For complexity reasons it is not best to iterate over everything that exists and since the predecessors and the successors are known it is not even necessary. To demonstrate the work with successors and predecessors, following function that goes through all the accessible nodes recursively can be defined:

```
def traverse(n, l = []):
    # input is node n and optional list l (of already
        traversed nodes)
    if n and n.ReferenceNode and n.msc and n.msc.HMsc
        and n.msc.start not in l:
        # Found new HMsc in reference node, traversing
        l.append(n.msc.start)
        traverse(n.msc.start, l)
    for node in n.succ:
        if node and node not in l:
            l.append(node)
            traverse(node, l)
        else:
            # We don't want to end up in any cycle
            # pass is just an empty body/statement
            pass
```

The algorithm above is defined for nodes but can easily be used for HMscs. It is only necessary to get start node of the HMsc and pass it to the function.

It is much easier and more algorithmic to use successors and predecessors to look through nodes. It might even bring better complexity to algorithms. Therefore function traverse and its modifications will most probably be used for high-level implementation when we'll be talking about paths in the future.

Definition 4.4 defines what is reachable node in a HMsc. It's node that is end of any existing path that begins in the HMsc's start node. Therefore (if we wish to optimize algorithm at least a little) to find whether node is reachable it is necessary to begin with start node of HMsc and try to find the node according to path rules (so called traversing). If we find it while traversing then result is True. Otherwise the node is not reachable, hence False. The additional code is here because of the optimization. We can easily use function traverse that will store every traversable node in its optional parameter l. The result might look like this:

```
def reachable(h, node):
    l = []
    # Lists are similar to pointers in C and can be
       changed inside of a function
    traverse(h.start, l)
    return node in l
```

Definition 4.5 is just simple usage of previous definitions and therefore can be transcribed quite easily.

```
def hasDeadlock(h):
    # Over all referenced HMsc's
    for hm in referenced(h):
        # Over all their nodes including start node
        for n in hm.snodes:
            if not n.ConnectionNode and not (n in h.
                nodes and n.EndNode):
                if reachable(h, n):
                    deadlock = True
                    for m in n.reachable:
                        if not m.ConnectionNode:
                            # We found reachable Node
                                that is not
                                ConnectionNode -> no
                                deadlock
                            deadlock = False
                    if deadlock:
                        return n
    return None
```

After the if statement it was assumed that there was deadlock but it was necessary to make sure that there was the deadlock with the for that followed. If any deadlock existed it was returned.

Using the basic 5 definitions, high-level code that suffices suggested notation could easily be written. Some definitions were used directly, some were not because they defined property but not the efficient way to find it.

4.3 Livelock checker

Livelock checker is quite similar to deadlock checker. It uses the same path notion when traversing the nodes. Specific cyclic traverser can be defined to detect all the cycles in the hmsc and look for a livelock.

```
def find_node_livelock(node, original, l = []):
    found = Set([]) # Set of HMscs that does not
        satisfy this condition
    if node.ReferenceNode and node.msc and node.msc.
        HMsc:
        # Found reference node that references HMsc
        found |= find_node_livelock(node.msc.start,
            original, l)
    for n in node.succ:
        if n not in l:
            # Node that was not traversed in this path
            l.append(n)
```

```
                found |= find_node_livelock(n, original, l
                    )
                l.pop()
            else:
                # Reached same node -> cycle
                cycle = l[l.index(n):]
                ref = False
                for i in cycle:
                    if i.ReferenceNode:
                        ref = True
                        if [x for x in i.reachable if x.
                            EndNode and x.owner == original
                            ]:
                            # Reachable EndNode was found
                            ref = False
                if ref:
                    found += n.owner
                    print("Livelock␣was␣found:␣", cycle +
                        [n])
    return found
```

The attribute original refers to the original HMsc and it is necessary for finding EndNode that belongs to the HMsc.

The traverser works almost the same way as the one for deadlock checker with an exception of pop after recursive call (and dealing with the livelock search but that is not a direct part of the traverser).

Cycle is quite easily detected (the same node is hit in the road) and extracted (everything from the place that node begins). The rest of the pseudocode looks for a cycle with a ReferenceNode that has no reachable EndNode.

The algorithm above defines function find_node_livelock that has node as an argument. Therefore following function that extends the algorithm to its final look must be defined:

```
def find_livelock(h, l = []):
    if h.HMsc:
        l.append(h.start)
        hmscs = find_node_livelock(h.start, h, l)
        l.pop()
    return hmscs
```

In the case of HMsc, the necessary node is obviously its StartNode. Therefore find_node_livelock is called for the start node of HMsc.

4.4 Acyclic checker

The definition of acyclic checker begins with the definition of visual order relation $<$. The definition can be treated as the constructive definition (and it is also expected). Therefore following function that returns visual order relation for a BMsc can be defined:

```
def buildVisOrder(bmsc):
    order = Set()
    for e1 in bmsc.events:
        for e2 in bmsc.events:
            if (e1 == e2) or (e1.area and e1.area.
               StrictOrderArea and e1.successor == e2)
                or ((e1, e2) in bmsc.
               CoregionEventRelation) or (e1.message
               and e1.message.CompleteMessage and e1.
               is_send and e1.message.receive_event ==
                e2) or (e1.area and e1.area.next == e2
               .area):
                order += (e1, e2)
    return order.transitive()
```

The function consists of the composed condition and transitive closure. It iterates over all the events in bmsc and inserts the events that satisfy given condition or are in its transitive closure (reflexive closure is part of the condition).

The result of the effort is the definition of acyclic BMsc so that function `is_acyclic` that tests whether computed visual order relation is antisymmetric visual order can be defined.

```
def is_acyclic(bmsc):
    visOrdRel = buildVisOrder(bmsc)
    # Test on antisymmetry
    for (i, j) in visOrdRel:
        if (j, i) in visOrdRel and i != j:
            return False
    return True
```

The extension for the HMsc is quite straightforward and can be defined quite easily.

```
def hmsc_is_not_acyclic(h):
    failed = Set()
    for h1 in referenced(h):
        for n in h1.nodes:
            if n and n.ReferenceNode and n.msc and n.
                msc.BMsc:
                print("Running␣is_acyclic")
                if not is_acyclic(n.msc):
                    failed += n.owner
    return failed
```

The function just iterates over all the nodes of all the referenced HMscs looking for a ReferenceNode that references BMsc. Once the node is found, regular is_acyclic function for BMsc is called.

4.5 FIFO checker

The definition of FIFO checker is based on the previous definitions and algorithms. The function itself just iterates over the events of a BMsc looking for two events satisfying given condition. The extension for HMsc is quite straightforward and very similar to the extension for acyclic checker. Implementation of the HMsc extension in the pseudocode can be found in fifo_checker.py.

```
def is_fifo(bmsc, chm):
    if not is_acyclic(bmsc):
        # BMsc is not acyclic, hence false
        return False
    visOrder = buildVisOrder(bmsc)
    for e1 in bmsc.events:
        for e2 in bmsc.events:
            if e1.is_receive and e2.is_receive and chm
                .same_channel(e1, e2) and (e1, e2) in
                visOrder and e1.message and e2.message
                and e1.message.CompleteMessage and e2.
                message.CompleteMessage and not ((e1.
                matching_event, e2.matching_event) in
                visOrder):
                return False
    return True
```

FIFO checker brings up another common aspect of checking algorithms - channel mapping (attribute chm).

Algorithm begins with the creation of visual order relation (as defined for acyclic checker) and checks whether the relation is acyclic. If it is acyclic all the events are iterated looking for two events in visual order relation that are CompleteMessages on same channel with matching events that are not in the appropriate visual order relation.

The condition could be easily satisfied by two events that share common Message (one of them as sender, the other one as receiver). Therefore it is necessary to check that the two events are of the same type (specifically receivers).

5 Integration to scstudio

The previous implementation of checking algorithms was written in C++ and is part of scstudio. The algorithms can be run from Microsoft Visio once a Msc is designed. The aim of this chapter is to describe how to integrate algorithms written in pseudocode into scstudio and to explain the way it is done.

5.1 Integrating algorithms

The algorithms are written in an extension of Python programming language but scstudio uses C++ data structures/classes. Checking algorithms in scstudio should define two new classes that are supposed to be derived from classes HMscChecker and BMscChecker. These two classes define virtual function check for HMsc and BMsc respectively. The function takes HMsc/BMsc pointer and returns list of HMsc/BMsc pointers (respectively) that do not satisfy the property.

Additionally the two new classes that should be defined must derive from class Checker that provides few basic pieces of information about the checker that will be implemented. Virtual functions get_property_name (returns name of the property), get_help_filename() (returns the location of help file), get_preconditions(), cleanup_attributes() (cleans attributes) and is_supported(chm) (true if the channel mapper chm is supported) should be defined. The definition of these functions is rather easy and does not differ from regular case when checking algorithms are written in C++.

In order to define function check (either for BMscChecker or HMscChecker) it is necessary to handle data structure written in C++ and provide a Python object for the Msc. The class PyConv was defined for this purpose. The class assumes that corresponding checkHMsc and checkBMsc functions are defined in pseudocode module. Both functions must take HMsc/BMsc and return list of HMscs/BMscs respectively. The class will abstract everything else.

To use the PyConv C++ class you need to create the class with the module name [1] as an argument and call checkHMsc or checkBMsc as you can see in this example:

```
HMscPtr hmsc;
BMscPtr bmsc;
```

1. Module name is the name of the file without its file extension (in this case without .py)

```
ChannelMapperPtr chm;
...
PyConv exp = PyConv("acyclic_checker"); // Initialize
    acyclic checker
exp.checkHMsc(hmsc, chm); // Run checker
exp.checkBMsc(bmsc, chm);
```

To simplify the creation of the two classes that are supposed to be written, bash script called gen_dir was created. The script includes file checker_list (that is in fact bash script containing only one variable) to get the list of checkers that are implemented in their high-level version and generates all the necessary files for integration with scstudio. The script generates files only if they do not exist yet. Therefore it is safe to modify generated cpp and h files (few things might need modification, for example path to documentation).

If you wish to integrate any new test all you need to do is to modify the variable CHECK in the checker_list file so that it contains your checker, implement functions checkHMsc and/or checkBMsc and add the file to the module where other checkers are located with respect to following naming convention:

- the name of the checker looks like regular word with leading capital letter (for example Deadlock, FIFO)
- the filename of test (module) is lower case version of the name of the checker with _checker appended to it (e.g. deadlock_checker, fifo_checker)
- the name of C++ classes that are generated are of the form PyXZChecker where X is H for HMsc and B for BMsc and Z is the name of the checker (e.g. PyHDeadlockChecker, PYBFIFOChecker)
- the name of the property is of the form PyXZFree where X can be H (HMsc) or B (BMsc) and Z is the name of the checker (e.g. PyBDeadlockFree, PyHFIFOFree)

In order to be able to see the checkers from Microsoft Visio few registry entries must be additionally created. This can be done in visio part of scstudio source code by following the example of other checkers in scstudio.

5.2 Data conversion

Since data must be shared between two different languages it must be converted between the two languages. Python offers API for this conversion.

All the standard objects can be converted into corresponding Python objects of type PyObject* (since everything in Python is object). The API also offers several functions to manipulate classes, call functions and basicly everything that can be done in standard Python interpreter.

The conversion is done within the checker function described in previous section and is handled by PyConv class.

This class consists of several functions for conversion of C++ data structures to Python objects. The class iterates the given BMsc or HMsc and creates and fills every corresponding structure that it finds during the iteration.

In the end when checking algorithm of the class returns list of HMscs or BMscs it will create new corresponding C++ structures and return them as a list.

The conversion and data handling between the two data models was abstracted and hidden from the users this way.

5.3 Universal checker

Previous sections described powerful tool for running checking algorithms written in executable pseudocode from developer point of view. The biggest concern is that a user must compile its own version of scstudio in order to add new checking algorithms to it.

The universal checker allows a user to just write a new checker in executable pseudocode and then run it from GUI without any need for compilation or registry modification.

This is done by introducing new Python package[2] for user defined checkers - pyscuser. This package is installed along with other Python extensions and is located in standard directory for Python extensions (usually PYTHON_LIBS/site-packages/pyscuser). The universal checker will then run all the properly named checking algorithms in this directory. There are actually two universal checkers - one for HMsc and one for BMsc.

If user wish to add a checking algorithm to this checker the user must name the file appropriately and place it in the directory of pyscuser package that was created purely for this purpose. The naming shall reflect the basic property of the checker. If the new checker implements function checkHMsc the name of the checker shall reflect it by placing capital H anywhere in it. If the new checker implements function checkBMsc the checker should have capital B in its name. Please note that checker must also have standard

2. Package in Python refers to a group of modules with initiator script. Modules from this package can be accessed as attributes of this package (by dot).

Python extension (.py) and cannot have two underscores at the beginning of its name (this is mostly due to standard Python convention where modules and variables starting with underscores are considered to be private or protected and are usually used for internal purposes of Python).

The universal checker is implemented as any other Python checker and therefore can be run from GUI as any other checker.

Input and output handling

The input that checkers get is converted from its C++ counterpart by PyConv class and is independent of the original data. Therefore modifications can be performed on the supplied HMsc or BMsc and the input can be returned by checkHMsc or checkBMsc in its modified form.

Once the property of checker is not satisfied the checker shall return a HMsc or BMsc that does not satisfy the property. The HMsc or BMsc can be modified without influencing its original C++ counterpart. Once a BMsc or HMsc is returned it is converted back to C++ and any other operations can be performed to the new C++ data structure.

Sometimes it might be useful to mark elements that do not satisfy the property of checker. The attribute marked was created for this purpose. The attribute's default value is False indicating that the element is not marked. If you wish to mark an element you just have to set the value of the attribute marked to True. When new HMsc or BMsc is converted from its C++ counterpart all the elements are unmarked.

6 Conclusion

In this thesis the notation for writing high-level checking algorithms was introduced. The notation was completely described and explained on examples. As was shown on examples the notation is simple and powerful enough.

The extension of Python's data model for checking algorithms was introduced, described and implemented. The extension is complete and allows comfortable and simple way to create message sequence charts by their textual form.

The direct extension of Python programming language - ineffective set and wrapper set were described and implemented. This contributed to the simplification of the notation quite significantly by hiding the necessity to deal with several set implementations.

Several checking algorithms were implemented to demonstrate functionality of implemented extensions. Basis for the implementation was written in pure mathematical notion. Although few modifications were introduced they correspond to a regular transformation of mathematical notion to pseudocode.

In order to further demonstrate functionality and simplify testing the integration to scstudio was implemented and described. Thanks to the supplied bash script integration is basically just a matter of modifying one line of one file and using suggested naming convention for files and functions (from developer point of view). Thanks to the supplied universal checker the integration is just a matter of copying one file to the directory of Python package that was created for this purpose and naming it properly (from user point of view).

The implementation could be further extended by addressing any of the existing extensions of MSC (for example time relations).

Bibliography

[1] *Scstudio : Sequence chart drawing and verification tool [online]. c2009 [cited 2011-01-01].* Available at: `http://scstudio.sourceforge.net/index.html`.

[2] *Gotthard, Petr - Babica, Jindřich - Řehák, Vojtěch. Sequence Chart Studio 0.1: Basic Verification Algorithms. 2008.*

[3] *The Python Language Reference [online]. c2010 [cited 2011-01-01].* Available at: `http://docs.python.org/py3k/reference/`.

[4] *Beazley, David. Python : referenční programátorská příručka. Praha : Neocortex, 2002.*

[5] *Jindřich Babica. Message Sequence Chart Properties and Checking Algorithms.* Master Thesis. Masaryk University, Brno, January 2009.

[6] *The GNU General Public Licence v3.0 [online]. c2010 [cited 2011-01-01].* Available at: `http://www.gnu.org/licenses/gpl-3.0.html`.

[7] *Standard operators as functions [online]. c2011 [cited 2011-01-03].* Available at: `http://docs.python.org/py3k/library/operator.html`.

[8] *Telecommunication Standardization Sector (ITU-T) [online]. c2011 [cited 2011-05-10].* Available at: `http://www.itu.int/ITU-T/`.

[9] *ITU Telecommunication Standardization Sector - Study group 17.* ITU recommendation Z.120, Message Sequence Charts (MSC), 2004.

[10] *What is an MSC? [online]. c2009 [cited 2011-05-10].* Available at: `http://www.sdl-forum.org/MSC/`.

[11] *Microsoft Visio [online]. c2010 [cited 2011-01-01].* Available at: `http://office.microsoft.com/en-us-/visio/`.