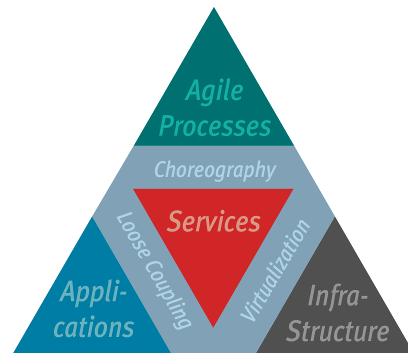
# Student Research Project Report

## LocalChecker Plugin for the jABC

July 2, 2007




**Institution**

University Dortmund
Department of Computer Science
Chair 5 of Programming Systems

**Supervisors**

Prof. Dr. Bernhard Steffen
Dipl. Inform. Ralf Nagel

**Trainee**

Johannes Neubauer

# Contents

**1 Introduction** 1

**2 Development** 2
- 2.1 Concept . . . 2
- 2.2 Plugin Interface . . . 2
- 2.3 Check Modes . . . 4
  - 2.3.1 Check of all SIBs . . . 4
  - 2.3.2 Continuous Checking . . . 4
- 2.4 LocalCheck Interface . . . 4
  - 2.4.1 Standard Checks . . . 5
- 2.5 Exception Handling . . . 7
- 2.6 Concurrency . . . 7
  - 2.6.1 Heavy Weight Task Manager . . . 7
  - 2.6.2 CheckSIBs . . . 7
  - 2.6.3 CheckObserver . . . 8

**3 Manual** 8
- 3.1 In the SIBGraph . . . 8
- 3.2 Plugin Inspector . . . 9
- 3.3 Info Window . . . 10

**4 Conclusion** 11

# 1 Introduction

Software engineering becomes more and more complex. *SOA*[1][6] is one strategy to cope with the growing scale of software systems. At Chair 5 for Programming Systems, Department for Computer Science at the University of Dortmund a tool called *jABC*[2][3] is under development which takes the idea of SOA on and enhances it. This programming or modeling paradigm is called lightweight process coordination. The application expert models graphs, composed of nodes called *SIBs*[3] and labeled edges called branches connecting the SIBs. A SIB graph model can be used from a first design to the point of a running system. Components of an application are represented by SIB classes, implemented by a programmer. The outgoing branches of a SIB are used as possible successors. The key idea is to integrate a new layer into the three-tier-architecture[5]. This layer is situated between the component and presentation layer and serves as a coordination tier.

SIB graphs are not constrained to executable models. The jABC consists of a robust core, called *Framework* which providing basic functionalities and the *Editor*, a GUI for the modeling process. More sophisticated features can be added with plugins. SIB graph models can be interpreted as *ER diagrams*[4] for the *DBSchema*[1] plugin, for example.

Furthermore the verification of properties of a SIB graph model, whilst the whole modeling process, is supported by the *model checking* plugin *GEAR*[5][2]. This concept does not afford the verification of local conditions to individual SIBs.

Therefore the task of this placement was to develope the *LocalChecker* plugin. It has to facilitate that local constraints concerning the SIB and its direct successors can be observed during the whole modeling process.

In short, the LocalChecker traverses a SIB graph model and executes the `checkSIB()` method in every SIB it reaches. This method contains testing code implemented by a SIB programmer called the SIB expert. In general the tests are constrained to the SIB instance and its direct successors. Verifying, whether a `URL` SIB parameter is reachable, would be a common test of a local check, for example. For every check that fails a message of a particular severity level is generated and attached to the SIB. These information can be accessed in the plugin inspector (see 3.2) or in the info window (see figure "Info Window" on page 10). Programmers of other plugins may access the messages via the `getUserObject()` method of the respective SIB or via `SIBUtilities.getMessage()`.

The next sections cover a short introduction to the LocalChecker, the design of the implementation and how to use it.

[1]**S**ervice **O**riented **A**rchitecture
[2]**J**ava **A**pplication **B**uilder **C**enter
[3]**S**ervice **I**ndependent building **B**lock
[4]Entity Relationship diagram
[5]Game-based, Easy And Reverse model-checking

# 2 Development

The LocalChecker plugin is closely connected to the jABC. The concept behind the plugin is illustrated in chapter "Concept" on the current page. The integration into the jABC is introduced in chapter "Plugin Interface" on this page. The plugin checks local properties of SIB instances, so a SIB has to implement some test code being executed when the instance is checked. This can be done by creating a SIB that implements the `LocalCheck` interface. Chapter "LocalCheck Interface" on page 4 describes the usage of this interface and how to use standard checks being provided by the plugin itself. Additionally chapter "Exception Handling" on page 7 describes the mechanisms of exception handling are described. Especially exceptions thrown in the check code have to be caught and communicated. Hence, error prone check code does not affect the robustness of the LocalChecker. Furthermore, the user is informed and can take care of the problem. In particular permanent checks (described in 2.3), but also complete checks with display of progress, need concurrency of the checking process. The solution for this problem is presented in chapter "Concurrency" on page 7.

## 2.1 Concept

The LocalChecker plugin does not check any conditions by itself, but provides the implementation of some standard checks. The actual checking code is up to the SIB implementor. This is, because the LocalChecker plugin cannot choose which checks are necessary for the respective SIB. Therefore a port defining how the test code can communicate the results of a check is inevitable. As LocalChecker enabled SIBs should be available even if the plugin is not present, the messages are simply added to the user objects of a SIB. This feature is supported by the JavaABC. More information on SIBs can be found in the documentation of the jABC[3]. The class `SIBUtilities` in the JavaABC Framework offers methods for adding and retrieving messages easily.

The LocalChecker gets a list of SIBs to test depending on the checking mode (see 2.3). An instance of the class `CheckSIBs` iterates through the elements. For every SIB, the messages of the last check are thrown away before the `checkSIB()` containing the check code (see 2.4) method of the SIB is executed. Afterwards the status of the SIB is evaluated. The status relates to the message with the highest severity during this check. If no message is present, the status will be set to *OK*. Figure 1 shows this procedure as an abstract SIB graph model.

## 2.2 Plugin Interface

Every plugin for the jABC has to implement the `Plugin` interface. The respective implementing class for the LocalChecker is `LocalCheckerPlugin`. The interface

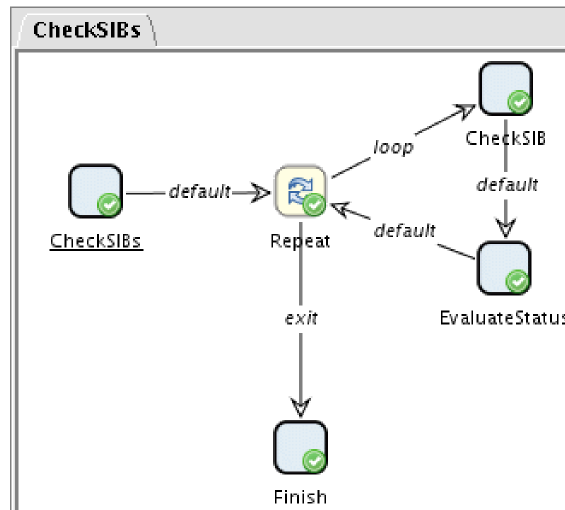



Figure 1: Checking Procedure

defines the following methods:

**getPluginName()** Returns the name of the plugin. It is displayed in the JavaABC Editor.

**getPluginInterface()** Returns the plugin interface. SIBs will be able to use plugin functionalities if they implement this interface. The respective class for the LocalChecker is `LocalCheck` (see 2.4).

**start()** Initializes the plugin. This procedure includes:

- adding some menu items to the plugin menu
- adding the plugin inspector to the inspector pane (see [3])
- attaching a model listener to the `GraphModelHandler` (see 2.3).

**stop()** Removes the components initialized in the `start()` method.

**getOverlayIcon()** Returns a small icon shown in a corner of the appropriate SIB icon in order to show the status of the respective plugin. It is possible to see the status of up to four plugins at a time. The LocalChecker returns the icon for the actual message level. The message level is as high as the message with the highest severity for that SIB. That means, that if there are info and warning messages, the status will be *warning* because a warning is more severe than an info message.

## 2.3 Check Modes

A check can be evoked by different actions. The plugin inspector and the plugin menu provide a button starting a complete check of the actual model. A toggle button activates and deactivates continuous checking.

### 2.3.1 Check of all SIBs

If continous checking is disabled, every change to a SIB will result in a status change to *unchecked*. This is done by the `LocalCheckerGraphListener` which is added to the `GraphModelHandler` during the initialization of the plugin. More information about the `GraphModelHandler` can be found in the developer documentation of the JavaABC[3]. The model listener is called when a model or SIB event has occured. If a check of all SIBs has been started, the class `CheckSIBs` is instantiated with a list of all SIBs of the current model.

### 2.3.2 Continuous Checking

If continuous checking is activated, the model listener will start a check with a list of all involved SIBs if an event is fired concerning the atual model or a SIB.

**Example 1** *Let two SIBs be connected with an edge. If the edge target is relocated to a third SIB by the user, a branch change event is fired. There are three SIBs that participate in this event, so they are checked.*

As a result, there are never SIBs with status *unchecked*. This feature needs to check SIBs while the user interacts with the GUI. Especially for long running checks this would freeze the Editor. Thus a framework for concurrency was built (see 2.6).

## 2.4 LocalCheck Interface

All SIBs that should be checked from the LocalChecker have to implement the `LocalCheck` interface. It defines the following methods:

**checkSIB()** contains the check code of the actual SIB.

**type()** returns an object representing the type of this SIB. This can be used in the `checksSIB()` method of the predecessors in order to check if this SIB is compatible to the predecessor.

The following example shows how a checking method might look like.

**Example 2**

```
public void checkSIB(SIB sib) {
  //Check incoming branches for their type.
```

```
  if(sib.getCell().getOutgoingBranches() != null) {
    for( SIBGraphEdge edge: sib.getCell().getIncomingBranches()) {
      SIB source = edge.getSourceCell().getSIB();
      if(source instanceof LocalCheck) {
        LocalCheck t = (LocalCheck)source;
        Object type = t.type();
        // If this SIB is of another type than TYPE_XY_SIB,
        // add an Info Message.
        if(type != null && !type.equals(TYPE_XY_SIB))
          SIBUtilities.addInfo(sib, "Parent SIB\"" +
              source.getName() + "\" is of type \"" +
              type.toString() \"". Expected \"" +
              TYPE_XY_SIB"\".);
      }
    }
  }
  //Check outgoing Branches if they are pointing to nirvana.
  if(sib.getCell().getOutgoingBranches() != null)
    for(SIBGraphEdge edge: sib.getCell().getOutgoingBranches())
      if(edge.getTargetCell() == null)
        SIBUtilities.addWarn(sib, "Outgoing Branch \"" +
           edge.getBranch() + "\" has no Target.");
}
```

*The example above is unnecessary, because these test cases are covered by the standard checks. Chapter 2.4.1 describes the already implemented tests and how they are used. Anyhow, the example shows that* `SIBUtilities` *offers methods like* `addInfo()` *for adding info messages to the user objects of a SIB.*

### 2.4.1 Standard Checks

Standard checks are already implemented, general checks that are delivered with the LocalChecker. The following checks are available at the time of writing:

**NO_EDGES** Checks whether the specified SIB has any incoming/outgoing edges. If the SIB is entirely disconnected from other SIBs, a warning is issued.

**UNASSIGNED_BRANCHES** Tests whether all branches of the specified SIB are considered by the current model. An info message is reported for every branch that is not assigned to an edge and not used as a model branch.

**UNASSIGNED_BRANCHES_ERROR** Checks whether all branches of the specified SIB are considered by the current model. An info is reported for every

branch that is not assigned to an edge and not used as a model branch. This method needs a string array of *errorbranches* as second parameter. For the *errorbranches* the check will produce an error message if the respective branch is unassigned.

**MISSING_BRANCH_LABELS** Verifies whether all outgoing branches have labels. This test produces warn messages.

**MISSING_SOURCES** Checks whether all incoming edges of the specified SIB have a source SIB. An error is reported for every incoming edge with a dangling start.

**MISSING_TARGETS** Checks whether all outgoing edges of the specified SIB have a target SIB. An error is reported for every outgoing edge with a dangling end.

**INCOMPATIBLE_SOURCES** Tests whether all source SIBs of the specified SIB have the given type. Warnings are generated for all source SIBs not matching the desired type.

**INCOMPATIBLE_TARGETS** Checks whether all target SIBs of the specified SIB have the given type. Warnings are generated for all target SIBs not matching the desired type.

**NOT_DOCUMENTED** Checks whether a SIB is documented. Reports info messages.

Every standard check has a second alternative called `NAMEOFCHECK_CUSTOM_LVL` expecting a second parameter setting the level of severity of messages generated by this test. It overwrites the default setting described above.

The following example shows, how a standard check can be used.

**Example 3**

```
import static ...StandardCheck.*;
// ...
  public void checkSIB(SIB sib) {
    EnumSet<StandardCheck> eset = EnumSet.of(INCOMPATIBLE_SOURCES,
        INCOMPATIBLE_TARGETS);
    for (StandardCheck check : eset)
      SIBUtilities.execute(check.getKey(), sib);
  }
```

*This method invokes the standard checks* `INCOMPATIBLE_SOURCES` *and* `INCOMPATIBLE_TARGETS`.

## 2.5 Exception Handling

Exceptions should not occur in check code. If an exception or error is thrown nevertheless, this has to be communicated because the programmer has to be able to correct the failure. This is done by catching every `Throwable` and generating a fatal message as a wrapper around the exception or error. This message will be handled as if it were a fatal error message from the `checkSIB()` method.

## 2.6 Concurrency

Local checks have to be executed in a concurrent thread to the *AWT Event Dispatcher Thread* in order to facilitate a non freezing GUI whilst modeling. This is achieved by a second queue similar to the *AWT Event Queue*. The result is called the *Heavy Weight Task Manager*.

### 2.6.1 Heavy Weight Task Manager

The Heavy Weight Task Manager processes `HWTasks` in a queued thread. The benefit of separating complex tasks from the Event Dispatcher Thread is a performance discharge for the GUI Event processing. In particular, tasks opening connections or waiting for some input from external sources would affect the latency of GUI processing. Especially continous checking is problematic in the case of the LocalChecker plugin.

This tool can be used for any time consuming task that should be run in a separate thread. The tasks are processed one after another, so there are no race conditions or deadlocks between two tasks. An intelligent implementation of the equals method in the `HWTask` can even prevent doubles in the task queue.

### 2.6.2 CheckSIBs

The LocalChecker uses the Heavy Weight Task Manager (HWTM) to process the local checks. Hence for continous checking, the checks are put into the queue to be executed one after another parallel to the GUI interactions of the user. For global checks the queue is cleared before the check is added to the queue.

The HWTM expects a `HWTask` for the `invokeLater()` and `invokeAndWait()` methods enqueueing a task either returning immediately after adding or waiting for the result. The class `CheckSIBs` implements the abstract class `AbstractHWTask`. This class can notify the observers when intermediate results are available. This is more convenient than implementing `HWTask` from scratch. `CheckSIBs` adds only SIBs that are not in the queue, so that no checks of a SIB are executed twice without a change of the SIB between the execution of the tests. Therefore the `equals()` method of `CheckSIBs` evaluates if the SIBs in its list are already in the queue. If no SIB is new to the queue, the task will not even be added.

### 2.6.3 CheckObserver

The HWTM offers the possibility to observe the progress of task execution in the queue. For this purpose an `Observer` can be added to the task manager which is notified when a new task is added, a task has finished or a task has intermediate results. The `CheckObserver` extends the class `HWTProgressMonitor` which opens a `ProgressMonitor` dialog (see [4]) in order to show the progress of a global check. The `HWTProgressMonitor` extends `Observer` and can be added to the task manager. Furthermore the status bar of the jABC shows the last checked SIBs.

# 3 Manual

The LocalChecker is responsible for testing local terms of SIBs. It visualizes messages about the results in the plugin inspector. There are four grades of messages denoting the severity of a problem.

**Message level:**

**Info** Clues of this grade have an informative character.

**Warning** Warnings remark that there might be a problem in the SIB. In general, they indicate that a parameter or branch deviates from the normal usage.

**Error** The SIB is used falsely. This can cause an unexpected behaviour of the graph.

**Fatal Error** The SIB is in an undefined state. The graph might be impaired by this severe error.

## 3.1 In the SIBGraph

All SIBs implementing the LocalChecker interface are highlighted with a small overlay icon, which marks the actual state. The overlay icons can be deactivated by the plugin menu or the respective button in the toolbar of the plugin inspector. Furthermore a position for the icons can be chosen in the options dialog in the plugins tab. Every LocalChecker enabled SIB is in one of six states.

**States of a SIB:**

**Ok** This SIB has been checked and there was nothing to report. Everything is fine.

**Info** There had been info messages only.

**Warning** At most warnings and info messages followed from the last check.

**Error** The last check produced at least one or more errors.

**Fatal Error** This SIB has fatal errors.

**Not Checked** The component was not checked since the last change.

The corresponding overlay icons are shown in figure 2. They are printed in the same order as the headwords above.

Figure 2: Overlay Icons

As "Highlight SIBs" is activated in the inspector, the overlay icon of every SIB is automatically updated when a check is performed or a SIB has altered.

## 3.2 Plugin Inspector

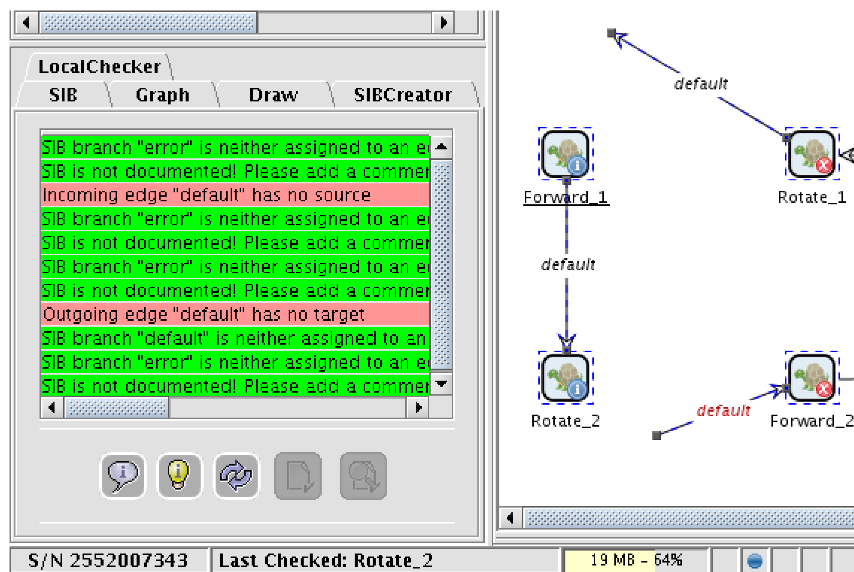



Figure 3: Plugin Inspector

The LocalChecker inspector (see figure 3) has a simple interface. It contains a table showing the messages of the actual selection in the `SIBGraph`. At the bottom of the inspector is a toolbar that enables the user to steer the plugin.

The button "ignore warnings" controls whether only errors and fatal errors are visible or warnings and infos, too. "Highlight SIBs" decides whether the overlay icons should be painted or not. If "continous checking" is activated, a SIB will be checked automatically if it has been changed. "Check all SIBs" checks the complete

SIBGraph and shows a dialog with statistics about the results. During the check a progress dialog provides status information. Generally a check is too fast for the dialog to pop up. In the case of a long running check, it can be canceled with the "cancel"-button in the progress dialog. The Button "info window" executes a test of all SIBs and opens a window with more detailed information about the results. The window contains the result of all SIBs, not only of the current selection as it is in the inspector. These buttons are available in the plugin menu, too.

## 3.3 Info Window

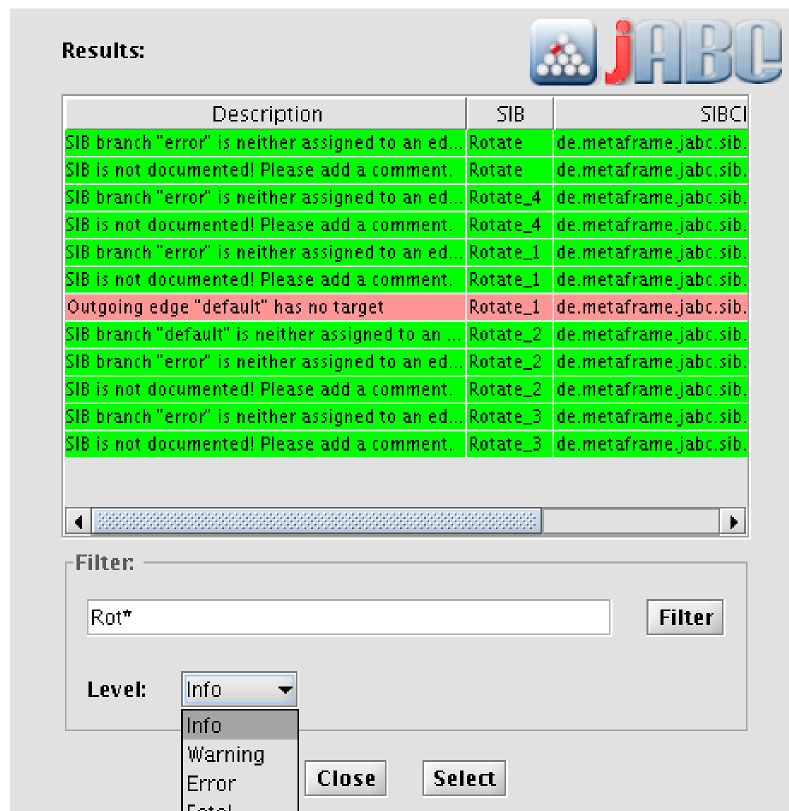



Figure 4: Info Window

The info window (see figure 4) presents the check results of all SIBs of the actual graph. The view can be constrained by filter methods. The filter text can embody wildcards[6]. All SIBs that do not match this pattern are not visible in the table. The "level filter" facilitates a more specific filtering of shown message levels than the "ignore warnings"-button of the inspector toolbar. By selecting the "error" level all warnings and infos are hidden, for example. The "select" button closes the

[6] "?" for an arbitrary character and "*" for an arbitrary character sequence

window and selects all SIBs in the SIBGraph owning messages that are selected in the result table of the info window. Clicking "ok" disposes the dialog without any further actions.

# 4 Conclusion

The LocalChecker plugin is a robust tool informing the application expert about incorrect usage of SIBs. In continous check mode, the plugin gives direct information about the status of a SIB via the overlay icon. For the actual selection, more details are available in the plugin inspector. The info window provides information about the status of all SIBs in a model. The LocalChecker is a powerful and easy to use addition to the model checking plugin GEAR.

# Index

AbstractHWTask, 7
AWT Event Dispatcher Thread, 7
AWT Event Queue, 7

CheckObserver, 8
CheckSIBs, 2, 4, 7

ER diagrams, 1
Exception Handling, 2, 7

GraphModelHandler, 3, 4
GUI, 1

Heavy Weight Task Manager, 7
HWTask, 7
HWTProgressMonitor, 8

jABC, 1
JavaABC Editor, 1, 3
JavaABC Framework, 1, 2

LocalCheck, 3, 4
LocalCheckerGraphListener, 4
LocalCheckerPlugin, 2

Model Checking, 1

Observer, 8

Plugin, 2
    DBSchema, 1
    GEAR, 1
    LocalChecker, 1
ProgressMonitor, 8

SIB, 1
SIBGraph, 9, 10
SIBUtilities, 1, 2, 5
SOA, 1
Standard Checks, 2

# References

[1] Christian Winkler. DBSchema. http://jabc.cs.uni-dortmund.de:8002/plugins/dbschema/index_en.html.

[2] Clemens Renner Marco Bakera. GEAR. http://jabc.cs.uni-dortmund.de:8002/plugins/gear_en.html.

[3] Ralf Nagel et al. Java ABC Framework Homepage. http://www.jabc.de.

[4] SUN. Java$^{TM}$ 2 platform standard edition 5.0 api specification. http://java.sun.com/j2se/1.5.0/docs/api/.

[5] Wikipedia. Multitier architecture. http://en.wikipedia.org/wiki/Three_layer_architecture.

[6] Wikipedia. Service Oriented Architecture. http://de.wikipedia.org/wiki/Serviceorientierte_Architektur.