Combining Monitoring with Run-time Assertion Checking

Gouw, C.P.T. de

Case Studies 5

In this chapter we use the formalism described in chapter Chapter 3 and the extension to design by contract described in chapter Chapter 4 to specify a Java library, and an industrial-sized case from the e-commerce company Fredhopper. The Java library we consider is a (last-in-first-out) Stack. The Stack example illustrates how the *Design by Contract* methodology as supported by JML can be used to specify the `push` and `pop`-methods purely in terms of histories in an elegant manner. In particular, this example shows how synthesized attributes of the start-symbol can be used conveniently inside method pre- and postconditions. Based on the case study, we discuss our experiences with SAGA.

5.1 Design by Contract: Stack

A Stack is an abstract data type which has only two operations `push` and `pop`. The operation `push` adds an object to the stack, while `pop` returns and removes the last element from the stack which was pushed but not yet removed. The operation `pop` is not allowed on an empty Stack. Figure 5.1 shows an interface for the Stack in Java.

```
public interface Stack {
       void push(Object item);
       Object pop();
}
```

Figure 5.1: Stack Interface

Our task is to find a specification for the Stack which ensures that `pop` is never called by the user on an empty stack, and moreover that `pop` returns the right object when called on a non-empty stack. The communication view in Figure 5.2 selects three events. The returns of push are needed to keep track of the elements which have been pushed onto the Stack. Note that it would incorrect be to consider the calls to `push` instead: suppose some strange implementation of `push` would itself call `pop` as its first action, before restoring the removed element and adding the element which was passed to `push`. Then calling `push` on an empty stack would fail (since that results in calling `pop` on an empty stack), but the history would be 'PUSH POP' (which seemingly looks valid for a Stack). Selecting returns of push avoids this problem. The calls to `pop`, which are referred to by the terminal 'POP', are needed to ensure that `pop` is never called on an empty Stack. In this case it would not suffice to track only returns of `pop`, since whenever `pop` is executed on an empty stack, the run-time checker would only detect the failure after executing of `pop` (which fails), and thus does not *prevent* unsafe behavior.

The protocol behavior of this view can be defined in terms of sequences of the *terminals* 'PUSH' and 'POP' generated by the context-free grammar given in Figure 5.3, where 's' is the

```
local view StackHistory specifies Stack {
       return push PUSH;
       call pop POP;
}
```

Figure 5.2: Communication View of a Stack

start symbol.

s	::=	PUSH s
	|	s s
	|	b
b	::=	PUSH b POP
	|	ϵ
	|	b b

Figure 5.3: Abstract Stack Behavior

The non-terminal 's' generates the *prefix closure* of the standard grammar for *balanced* sequences of 'PUSH' and 'POP' (which are generated by the non-terminal 'b'). This ensures that `pop` is never called on an empty stack.

In order to specify the relation between the actual parameters of calls to the `push` method and the return values of the `pop` method, we introduce a synthesized attribute 'stack' of type `JMLListValueNode` for the non-terminal 's'. `JMLListValueNode` is a JML class for a singly-linked list with *side-effect free* implementations of the methods `JMLListValueNode append(Object item)`, which appends an item to the list, and `JMLListValueNode concat(JMLListValueNode ls2)` which concatenates two lists. The intended value of the 'stack' attribute is a list of the elements which are pushed but have not yet been popped. Since balanced Stacks are empty, associating the 'stack' attribute also to the b-non-terminal would be redundant. Figure 5.4 shows how 'stack' is updated in each production of the non-terminal s. Intuitively the value of 'stack' at the root of the parse-tree (i.e. an occurence of the start-symbol s) is a list containing the current contents of the

Stack. Figure 5.5 shows the parse tree for the history resulting from the program `s.push(5); s.push(7); s.pop();`. Note that this does not mean that an actual implementation of the stack interface works correctly: the attribute grammar can be considered as a 'reference implementation' of the stack, but we still need to ensure that an actual implementation of the Stack matches (in the sense that calling `pop` returns the right value) this reference implementation.

s	::=	PUSH s_1	(`stack =`s_1`.stack.append(PUSH.item);`)
	|	s_1 s_2	(`stack =`s_1`.stack.append(`s_2`.Stack);`)
	|	b	(`stack = stack.clear();`)
b	::=	PUSH b POP	
	|	ϵ	
	|	b b	

Figure 5.4: Attribute Grammar Stack Behavior

In order to specify the method contracts for the Stack, the JML implementation of SAGA (described in Section 4.1) allows referring to the synthesized attributes of the root of the parse tree. Since the start symbol in the parse tree gener-

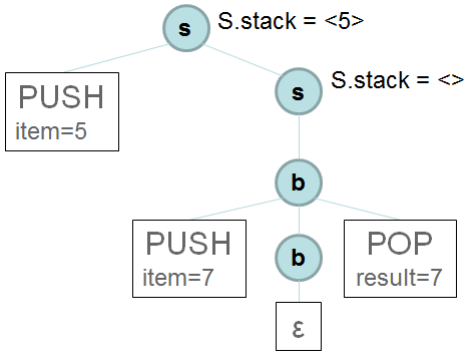

Figure 5.5: Parse tree annotated with attribute values for the history push(5) push(7) pop() in the grammar of Figure 5.4 (irrelevant attributes ommitted)

ates the whole history, intuitively the synthesized attributes of the start symbol can be thought of as a property of the entire history. In order to use the attribute 'stack' of this grammar in assertions for specifying the contracts of the `push` and `pop` methods of the 'Stack' interface (Figure 5.1) in terms of communication histories, the modeling framework provides a class `StackHistory` which corresponds to the communication view of Figure 5.2. This class contains a 'getter' method `JMLListValueNode stack()` which retrieves the value of the attribute 'stack' of the root of the parse tree of the current history.

```
interface Stack {
  //@ public model instance StackHistory history;

  //@ ensures history.stack().equals(
  //@           \old(history.stack()).append(item));
  void push(Object item);

  //@ ensures history.stack().equals(
  //@           \old(history.stack()).tail());
  //@ ensures \result == \old(history.stack()).head();
  Object pop();
}
```

Figure 5.6: JML Specification Stack Interface

Figure 5.6 illustrates how the `StackHistory` class can be used to specify the desired contracts. The JML keyword `model` indicates that `history` (of type `StackHistory`) can be used only in specifications. The keyword `instance` specifies that `history` will be added as a (non-static) field to any class that implements the `Stack` interface. The `ensures` and `requires` clauses specify the method contracts in terms of the 'stack' attribute (whose value is defined in the attribute grammar). Summarizingly, the property that `pop` may not be called on an empty stack is ensured by the productions of the grammar (the grammar productions can be considered to be an interface invariant for the protocol behavior), and the property that `pop` returns the right object is guaranteed by the method contracts and the definition of the attribute 'stack'.

Note that alternatively we could have avoided the method contracts by instead adding appropriate assertions in the attribute grammar before and after *every* occurence of 'PUSH' and 'POP' in the grammar. This leads to duplication since 'PUSH' occurs multiple times in the grammar. Moreover, for this alternative solution, we should also have added to the communication view that we intend to capture returns of `pop`: otherwise there would be no way to check that `pop` returned the right value. For the above example, we favour the above design-by-contract solution over the assertions-in-grammar, since it avoids duplication of specifications and additionally avoids adding the extra terminal for returns of `pop`. This increases readability of the grammar, and results in less overhead for the run-time check since the sequence of tokens to parse is shorter.

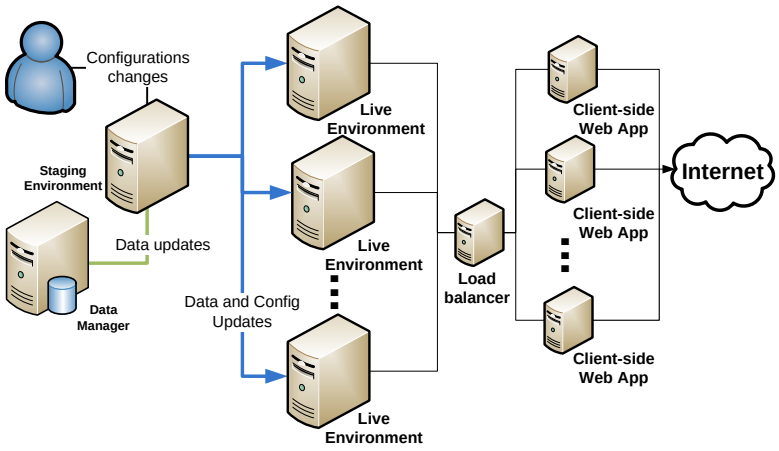

Figure 5.7: An example FAS deployment

5.2 Fredhopper Case-Study

Fredhopper[1] is a search, merchandising and personalization solution provider, whose products are tailored to the needs of online businesses. Fredhopper operates behind the scenes of more than 100 of the largest online shops [2]. It provides the Fredhopper Access Server (FAS), which is a distributed concurrent object-oriented system that provides search and merchandising services to eCommerce companies. Briefly, FAS provides to its clients structured search capabilities within the client's data. Each FAS installation is deployed to a customer according to the FAS deployment architecture (See Figure 5.7).

FAS consists of a set of live environments and a single staging environment. A live environment processes queries from client web applications via web services. FAS aims at providing a constant query capacity to client-side web applications. A staging environment is responsible for receiving data updates in XML format, indexing the XML, and distributing the resulting indices across all live environments according to the *Replica-*

[1] `http://www.sdl.com/products/fredhopper/`

[2] `http://www.sdl.com/campaign/wcm/gartner-maqic-quadrant-wcm-2013.html?campaignid=70160000000fSXu`

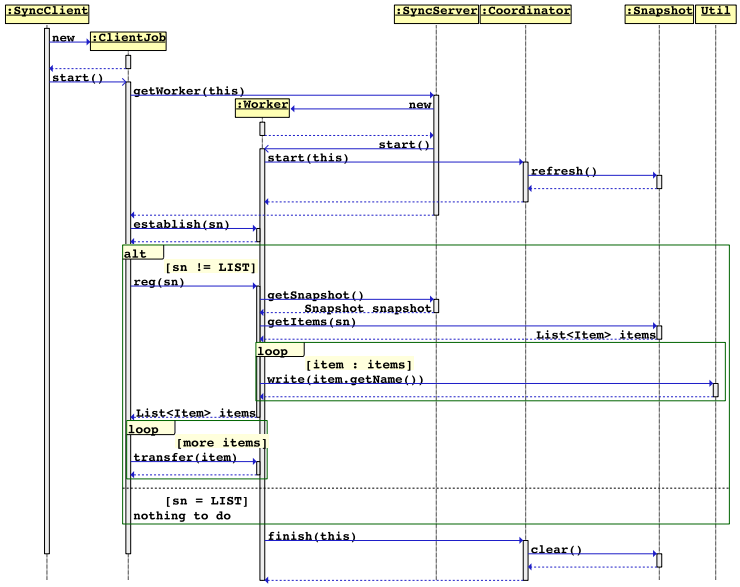

Figure 5.8: Replication interaction

tion Protocol. The Replication Protocol is implemented by the *Replication System*. The Replication System consists of a *SyncServer* at the staging environment and one *SyncClient* for each live environment. The SyncServer determines the schedule of replication, as well as its content, while SyncClient receives data and configuration updates according to the schedule.

Replication Protocol

The SyncServer communicates to SyncClients by creating *Worker* objects. Workers serve as the interface to the server-side of the Replication Protocol. On the other hand, SyncClients schedule and create *ClientJob* objects to handle communications to the client-side of the Replication Protocol. When transferring data between the staging and the live environments, it is important that the data remains *immutable*. To ensure immutability without interfering the read and write accesses of the staging environment's underlying file system,

```
interface Snapshot {
 void refresh();
 void clear();
 List<Item> items(String sn);
}

interface Worker {
 void establish(String sn);
 List<Item> reg(String sn);
 void transfer(Item item);
 SyncServer server();
}
```

Figure 5.9: SnapShot and Worker interfaces of Replication System

the SyncServer creates a *Snapshot* object that encapsulates a snapshot of the necessary part of the staging environment's file system, and periodically *refreshes* it against the file system. This ensures that data remains immutable until it is deemed safe to modify it. The SyncServer uses a *Coordinator* object to determine the safe state in which the Snapshot can be refreshed. Figure 5.8 shows a UML sequence diagram concerning parts of the replication protocol with the interaction between a SyncClient, a ClientJob, a Worker, a SyncServer, a Coordinator and a Snapshot. the diagram also shows a *Util* class that provides static methods for writing to and reading from `Stream`. The figure assumes that SyncClient has already established connection with a SyncServer and shows how a ClientJob from the SyncClient and a Worker from a SyncServer are instantiated for interaction. For the purpose of this paper we consider this part of the Replication Protocol as a *replication session.*

In this section we show how to modularly decompose object interaction behavior depicted by the UML sequence diagram in Figure 5.8 using SAGA. Figures 5.9 and 5.10 shows the corresponding interfaces and classes, note that we do not consider SyncClient as our interest is in object interactions of a replication session, that is after `ClientJob.start()` has been invoked.

The protocol descriptions and specifications considered in this case study have been obtained by manually examining the

```
interface SyncServer {
 Snapshot snapshot();
}

interface Coordinator {
 void start(Worker t);
 void finish(Worker t);
}

class Util {
 static void write(String s) { .. }
}
```

Figure 5.10: SyncServer and Coordinator interfaces of Replication System

behavior of the existing implementation, by formalizing available informal documentations, and by consulting existing developers on intended behavior. Here we first provide such informal descriptions of the relevant object interactions:

- Snapshot: at the initialization of the Replication System, `refresh` should be called first to refresh the snapshot. Subsequently the invocations of methods `refresh` and `clear` should alternate.
- Coordinator: neither of methods `start` and `finish` may be invoked twice in a row with the same argument, and method `start` must be invoked before `finish` with the same argument can be invoked.
- Worker: `establish` must be called first. Furthermore `reg` may be called *if* the input argument of `establish` is not "LIST" but the name of a specific replication schedule, and that `reg` must take that name as an input argument. When the `reg` method is invoked and before the method returns, the Worker must obtain the replication items for that specific replication schedule via method `items` of the Snapshot object. The Snapshot object must be obtained via method `snapshot` of its SyncServer, which must be obtained via the method `server`. It must notify the name of each replication item to its

```
local view SnapshotHistory
grammar Snapshot.g
specifies Snapshot {
 call void refresh() rf,
 call void clear() cl
}
```

Figure 5.11: Snapshot Communication View

```
local view CoordinatorHistory
grammar Coordinator.g
specifies Coordinator {
 call void start(Worker t) st,
 call void finish(Worker t) fn
}
```

Figure 5.12: Coordinator Communication View

```
local view WorkerHistory grammar Worker.g
specifies Worker {
 call void establish(String sn) et,
 call List<Item> reg(String sn) rg,
 return List<Item> reg(String sn) is,
 call void transfer(Item item) tr
}
```

Figure 5.13: Worker Communication View

interacting SyncClient. This notification behavior is implemented by the static method `write` of the class `Util`. The method `reg` also checks for the validity of each replication item and so the method must return a subset of the items provided by the method `items`. Finally `transfer` may be invoked after `reg`, one or more times, each time with a unique replication item, of type `Item`, from the list of replication items, of type `List<Item>`, returned from `reg`.

Figures 5.11 to 5.14 specifies communication views. They provide partial mappings from message types (method calls and returns) that are local to individual objects to grammar terminal symbols. Note that the specification of the Worker's behavior is modularly captured by two views: `WorkerHistory` and

```
local view WorkerRegHistory grammar WorkerReg.g
specifies Worker {
 call List<Item> reg(String sn) rg,
 return List<Item> reg(String sn) is,
 return Snapshot SyncServer.snapshot() sp,
 call List<Item> Snapshot.items(String sn) ls,
 return List<Item Snapshot.items(String sn) li,
 call static void Util.write(String s) wr
}
```

Figure 5.14: WorkerReg Communication View

$$\begin{array}{lcll} S & ::= & \epsilon & \mid rf\ T \\ T & ::= & \epsilon & \mid cl\ S \end{array}$$

Figure 5.15: Snapshot Attribute Grammar

$$\begin{array}{lcll} S & ::= & & T \quad (T\texttt{.ts} = \texttt{new HashSet();}) \\ T & ::= \epsilon & \mid & st \quad \{\texttt{assert !}\ T\texttt{.ts.contains(}st\texttt{.t);}\} \\ & & & (T\texttt{.ts.add(}st\texttt{.t);})\ T_1\ (T_1\texttt{.ts} = T\texttt{.ts;}) \\ & & \mid & fn \quad \{\texttt{assert}\ T\texttt{.ts.contains(}fn\texttt{.t);}\} \\ & & & (T\texttt{.ts.remove(}fn\texttt{.t);})\ T_1\ (T_1\texttt{.ts} = T\texttt{.ts;}) \end{array}$$

Figure 5.16: Coordinator Attribute Grammar

`WorkerRegHistory`. The view `WorkerHistory` exposes methods `establish`, `reg` and `transfer`. Using this view we would like to capture the overall valid interaction in which Worker is the callee of methods, and at the same time the view helps abstracting away the implementation detail of individual meth-

$$\begin{array}{lcll} S & ::= \epsilon & \mid & et\ T \quad (T\texttt{.d} = et\texttt{.sn;}) \\ T & ::= \epsilon & \mid & \{\texttt{!"LIST".equals(}T\texttt{.d);}\}? \\ & & & rg\ \{\texttt{assert}\ rg\texttt{.sn.equals(}T\texttt{.d);}\}\ U \\ U & ::= \epsilon & \mid & is\ V \quad (V\texttt{.m} = \texttt{new ArrayDeque(}is\texttt{.result);}) \\ V & ::= \epsilon & \mid & tr \quad \{\texttt{assert}\ V\texttt{.m.peek().equals(}tr\texttt{.item);}\} \\ & & & (V\texttt{.m.pop();})\ V_1\ (V_1\texttt{.m} = V\texttt{.m;}) \end{array}$$

Figure 5.17: Worker Attribute Grammar

```
/*S accepts call to Worker.reg() and, records */
/*the input schedule name, also S allows */
/*arbitary calls to SyncServer.snapshot() */
/*and Util.write() */
S   ::= ϵ  |  wr S | sp S | rg T   (T.d = et.sn;)

/*T accepts and stores the return */
/*snapshot object from SyncServer.snapshot() */
T   ::= ϵ  |  sp V (V.d = T.d; U.s = sp.result;)

/*U ensures call items() is called on the same */
/*snapshot object, and the replication items */
/*for the correct schedule are retrieved */
U   ::= ϵ  |  ls {assert ls.callee.equals(U.s);
                 assert ls.sn.equals(U.d);}
              V (V.s = U.s;)

/*V records replication items and their name */
/*returned from item() */
V   ::= ϵ  |  li W (W.is = new HashSet(li.result);
                    W.ns = new HashSet();
                    for (Item i :W.is) {
                      W.ns.add(i.name()); })

/*W ensures all replication */
/*items are processed */
W   ::= ϵ  |  wr (W.ns.remove(wr.s);)
              W1 (W1.ns =W.ns; W1.is =W.is;)
           |  is {assert W.is.containsAll(is.result);
                 assert W.ns.isEmpty();}
              X

X   ::= ϵ  |  sp X | rg X
```

Figure 5.18: WorkerReg Attribute Grammar

ods. The view `WorkerRegHistory`, on the other hand, captures the behavior inside `reg`. According to the informal description above, the view projects incoming method calls and returns of `reg`, outgoing method calls to `server` and `items`, and as well as the outgoing static method calls to `write`.

We now define the abstract behavior of the communication views, that is, the set of allowable sequences of interactions of objects restricted to those method calls and returns mapped in the views. Each local view also defines the file containing the attribute grammar, whoses terminal symbols the view maps method invocations and returns to. Specifically, Figures 5.15 to 5.18 shows the attribute grammars `Snapshot.g`, `Coordinator.g`, `Worker.g` and `WorkerReg.g` for views `SnapshotHistory`, `CoordinatorHistory`, `WorkerHistory` and `WorkerRegHistory` respectively.

The simplest grammar `Snapshot.g` specifies the interaction protocol of Snapshot. It focuses on invocations of methods `refresh` and `clear` per Snapshot object. The grammar essentially specifies the (prefix-closure of the) regular expression (`refresh clear`)$*$.

The grammar `Coordinator.g` specifies the interaction protocol of Coordinator. It focuses on invocations of methods `start` and `finish`, both of which take a Worker object as the input parameter. These method calls are mapped to terminal symbols st and fn, while their inherited attribute is a `HashSet`, recording the input parameters, thereby enforcing that for each unique Worker object as an input parameter only the set of sequences of method invocations defined by the requlars expression (`start finish`)$*$ is allowed.

The grammar `Worker.g` specifies the interaction protocol of Worker It focuses on invocations and returns of methods `establish`, `reg` and `transfer`. The grammar specifies that for each Worker object, `establish` must be first invoked, then followed by `reg` and then zero or more `transfer`, that is, the regular expression (`establish reg transfer`$*$). We use the attribute definition of the grammar to ensure the following:

- The input argument of `establish` and `reg` must be the same;

- `reg` can only be invoked if the input argument of `establish` is not "LIST";
- The return value of `reg` is a list of `Item` objects such that `transfer` is invoked with each of `Item` in that list from position 0 to the size of that list.

The grammar `WorkerReg.g` specifies the behavior of the method `reg` of Worker. It focuses on the invocations and returns of method `reg` of Worker as well as the outgoing method calls and returns of `Util.write` and `SyncServer.snapshot` and `Snapshot.items`. At the protocol level the grammar specifies the regular expression (`snapshot items write`$*$) inside the invocation method `reg`. We use attribute definition to ensure the following:

- `Snapshot.items` must be called with the input argument of `reg` and it must be called on the Snapshot object that is identical to the return value of `SyncServer.snapshot`;
- The static method `Util.write` must be invoked with the value of `Item.name` for each `Item` object in the Collection returned from `Snapshot.items`;
- The returned list of `Item` objects from `reg` must be a subset of that returned from `Snapshot.items`.

Notice that methods `Util.write` and `SyncServer.snapshot` may be invoked outside of the method `reg`. However, this particular behavioral property does not specify the protocol for those invocations. The grammar therefore abstracts from these invocations by allowing any number of calls to `Util.write` and `SyncServer.snapshot` before and after `reg`.

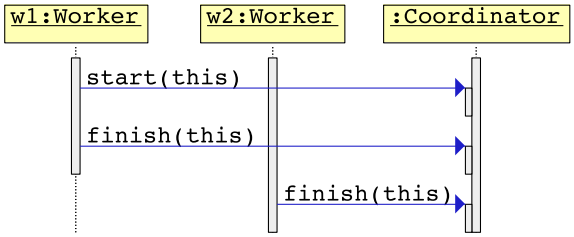

Figure 5.19: Violating histories

5.3 Experiment

We applied SAGA to the Replication System. The current Java implementation of FAS has over 150,000 lines of code, and the Replication System has approximately 6400 lines of code, 44 classes and 5 interfaces.

We have successfully integrated SAGA into the quality assurance process at Fredhopper. The quality assurance process includes automated testing that includes automated unit, integration and system tests as well as manual acceptance tests. In particular system tests are executed twice a day on instances of FAS on a server farm. Two types of system tests are scenario and functional testing. Scenario testing executes a set of programs that emulate a user and interact with the system in predefined sequences of steps (scenarios). At each step they perform a configuration change or a query to FAS, make assertions about the response from the query, etc. Functional testing executes sequences of queries, where each query-response pair is used to decide on the next query and the assertion to make about the response. Both types of tests require a running FAS instance and as a result we may leverage SAGA by augmenting these two automated test facilities with runtime assertion checking using SAGA.

To integrate of SAGA with the system tests, we employ Apache Maven tool[3], an open source Java based tool for managing dependencies between applications and for building dependency artifacts. Maven consists of a project object model

[3] `maven.apache.org`

```
class WKImpl extends Thread
implements Worker {
 final Coordinator c;
 WKImpl(Coordinator c) {
   this.c = c; }
 public void run() {
  try { .. c.start(this); ..
  } finally {
    c.finish(this); .. }}}
```

Figure 5.20: Incorrect behavior of `WKImpl`

(POM), a set of standards, a project lifecycle, and an extensible dependency management and build system via plug-ins. We use its build system to automatically generate and package the parser/lexer of attribute grammars as well as aspects from views and grammars. We expose the packaged aspects, parser and lexer to FAS instance on the server farm and employ Aspectj using load-time weaver for monitoring method calls/returns during the execution of FAS instances on the server farm. Table 5.1 shows the number of join point matches during the execution of 766 replication sessions over live client data. Figure 5.21 shows the exection time of the 766 replication sessions with and without the integration of SAGA in milliseconds. At some points (for example, around 261 events), the figure seemingly indicates that the system runs faster with SAGA than without. In reality this is not the case: the dependence of the case study on user input (i.e., to start replication sessions) means that it is impossible to replicate an execution exactly (with the only difference being SAGA turned on and off respectively) and leads to small errors in the measurements. However, despite the fact that we cannot control the exact flow of control of the replication sessions (due to this dependence on user input), the graph clearly shows that the integration of SAGA has minimal performance impact on the execution time.

During this session we have found an assertion error at join point `call finish` due to the condition T`.ts.contains(`fn`.t)` not being satisfied at non-terminal T of the grammar Coordinator.g. Specifically, the implementation of Worker (`WKImpl`) that invoke `finish` before `start`. Figure 5.19 shows the sequence diagram of an invalid history

Join point	Terminal	Match
`call static write`	wr	247446
`return snapshot`	sp	3061
`call transferItem`	tr	1101
`return reg (WorkerHistory)`	is	765
`return reg (WorkerRegHistory)`	is	765
`call establish`	et	766
`call reg (WorkerHistory)`	rg	765
`call reg (WorkerRegHistory)`	rg	765
`return items`	li	765
`call start`	st	766
`call finish`	fn	766
`call items`	ls	765
`call refresh`	rf	766
`call clear`	cl	766

Table 5.1: Join point matches in 766 replication sessions

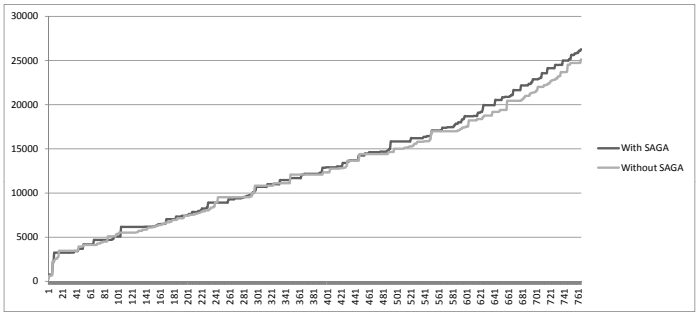

Figure 5.21: Comparison of the execution time (milliseconds) of the replication sessions with and without the integration of SAGA

causing the error, fully automatically generated from the output of SAGA. Figure 5.20 shows part of the implementation of `WKImpl`. It turns out that in the `run` method of `WKImpl`, the method `start` is invoked inside a `try` block while the method `finish` is invoked in the corresponding `finally` block. As a result when there is an exception being thrown by the execu-

tion preceding the invocation of `start` inside the `try` block, for example a network disruption, `finish` would be invoked without `start` being invoked.