Scalable SD Erlang Reliability Model

Natalia Chechina, Huiqing Li,
Phil Trinder, and Simon Thompson

School of Computing Science
The University of Glasgow
Glasgow G12 8QQ
UK

Technical Report TR-2014-004

December 23, 2014

Abstract

This technical report presents the work we have conducted to support SD Erlang reliability and to formally specify the semantics of s_groups. We have considered the following aspects of SD Erlang reliability: node recovery after failures and s_group name uniqueness.

The first aspect is restarting nodes in their s_groups after failures (Chapter 2). We have considered a number of approaches to implement node restart, and have implemented one of them. The mechanism is optional, and records s_group state information for nodes which are started with the `-rsconfig` flag.

The second aspect considers s_groups with the same name (Chapter 3). In SD Erlang we do not guarantee s_group name uniqueness; however, we guarantee a high probability of s_group name uniqueness when s_groups are started dynamically – for that we have modified function `s_group:new_s_group`. When s_groups are started at launch the configuration file is checked to minimise occurrence of s_groups with the same name. We have also ensured correct behaviour of s_groups and nodes in case duplicate s_group names occur.

We introduce an operational semantics for SD Erlang s_groups that defines an abstract state and presents transition of fifteen SD Erlang functions (Chapter 4). We have conducted a correctness check of the implemented functions using QuickCheck [AHJW06] testing tool (Chapter 5).

The reliable SD Erlang s_groups are open source and can be found in Github `https://github.com/release-project/otp/tree/17.4-rebased`.

Contents

Chapter 1

Introduction

The objectives of this technical report are to implement and validate a scalable reliability model for SD Erlang by minimising supervision interconnections.

SD Erlang reliability includes a wide spectrum of issues, such as node recovery after failures, correctness of Erlang node behaviour in s_groups, and correctness of implemented functions. We start with a discussion on node recovery after failures in Chapter 2. Here, we cover a wide range of possible node recovering mechanisms, and implement one of them in SD Erlang. Then we discuss s_groups with the same name in Chapter 3. We started checking the correctness of the functions in [CLTG14] by conducting an extensive unit testing. In this technical report we introduce SD Erlang semantics. We describe fifteen SD Erlang functions in mathematical terms using state transition semantics in Chapter 4. Then we compare results of mathematical calculations and of the real implementation using QuickCheck [AHJW06] testing tool in Chapter 5.

Chapter 2

Restarting a Failed Node

The mechanism for restarting a failed node consists of three main components: collecting s_group state information of the node, detecting the node failure, and restarting the node using the s_group state information. In this section we first discuss different strategies to collect node s_group information (Section 2.1). Then we consider strategies that would be useful in the benchmarks we use in the RELEASE project (Section 2.2), and finally we present the strategy that we have implemented in SD Erlang (Section 2.3).

2.1 Recovering S_group State

A node may join or leave s_groups, so we need to record s_group membership information to correctly restart a failed node.

The information collection mechanism can be implemented in a number of ways. To explore different alternatives we use system state estimation, ξ, that analyses *where* state information is collected, *with whom* it is shared, *who* initiates the exchange and *how often* it happens [Rot94]. More formally,

$$\xi \in \{Centralised, Decentralised, Hybrid\} \times \{Complete, Partial, Variable\} \times$$
$$\times \{Voluntary, Involuntary, Composite\} \times \{Periodic, Aperiodic, Combination\} \quad (2.1)$$

Below we discuss the meaning of the parameters and examples of how a particular property can be implemented to collect s_group information.

The first set *(Centralised, Decentralised, Hybrid)* defines *where* s_group state information is collected. In case of a centralised scheme a dedicated node or a process may collect s_group state information from the whole system. A decentralised scheme implies that each node is responsible for storing its own s_group state information, for example, in a `.config` file. A hybrid scheme combines both centralised and decentralised schemes, e.g. an s_group leader node collects s_group state information from the members of its s_group in a centralised manner, and then s_group leader nodes exchange the state information between each other in a decentralised manner.

The second set *(Complete, Partial, Variable)* defines *with whom* s_group state information is shared. A complete scheme implies that every process that collects s_group state information has information about all nodes of the system. In a partial scheme an information collecting process has only information about the nodes it collects information from. A variable scheme means that some processes have partial information and some processes have full information. The scheme is suitable for the cases when information is collected in a hierarchical manner, e.g. processes of *Level*1 collect information from the nodes and pass it to processes of *Level*2. Thus, *Level*1 processes have up-to-date partial information; whereas *Level*2 processes have information about all nodes in the system, but this information may be out-of-date.

The third set *(Voluntary, Involuntary, Composite)* defines *who* initiates the information exchange. In a voluntary scheme nodes initiate sending s_group state information to the collecting process themselves. In an involuntary scheme nodes send s_group state information as a response on a request to provide the information. In a composite scheme some information is send voluntary and some is sent involuntary. For example, nodes send their s_group information to *Level*1 processes voluntary, whereas *Level*1 processes send information to *Level*2 processes involuntary.

The fourth set *(Periodic, Aperiodic, Combination)* defines *how often* the information exchange happens. The periodic and aperiodic schemes are self explanatory, for example, information is exchanged every 30 sec or every time the node s_group state changes. In a composite scheme nodes may send s_group state information to *Level*1 processes aperiodically, and *Level*1 processes send information to *Level*2 processes periodically.

The system state estimation analysis shows that there are many ways that the s_group information collecting algorithm can be implemented. Each algorithm has target applications and some of these algorithms are more complicated than others. To identify the most suitable type of the node restarting algorithm for SD Erlang we analyse exemplars we work with in the RELEASE project [BCC+12] (Section 2.2).

2.2 Exemplars

In this section we consider four exemplars we currently use in the RELEASE project to evaluate distributed Erlang and SD Erlang, i.e. Sim-Diasca, Riak, Orbit, and Mandelbrot set. The exemplars are representatives of typical distributed Erlang applications. All benchmarks are open source. The benchmark details and their possible implementations in SD Erlang are discussed in [CTG+14]. Therefore, in this section for every benchmark we only discuss principles of grouping nodes in s_groups and nodes dynamic membership of s_groups.

Sim-Diasca is a simulation engine developed in EDF [EDF10]. When new processes are spawned the aim is to place often communicating processes close to each other. Thus, nodes can be grouped on the basis of locality. Currently, the engine executes on a fixed number of nodes, and s_group configuration will not change much during the program execution. Therefore, nodes can be grouped in s_groups either at the node launch or dynamically before the actual simulation has started.

Riak is a database management system developed in Basho [Klo10]. The Riak's aim is to provide access to the stored data even when some nodes have failed. Nodes can be grouped in s_groups on the basis of Riak preference lists, i.e. a node forms an s_group with the nodes on which its replica data is kept. Thus, each node will have its own s_group. When a node fails its s_group information will be redistributed and the s_group will be deleted. When a node gets back again it will join s_groups of the neighbouring nodes and create its own s_group. The node will know its neighbours as they will support its restart; from the neighbours the node will know which nodes to add to its own s_group and which s_groups to join. Thus, Riak nodes will join and leave s_groups dynamically, and essential s_group information will be provided by peer nodes.

Orbit is a computation oriented, distributed hash table benchmark. A hash table that collects results of calculations is partitioned between the nodes. The aim of the application is to explore a given set of values using a set of generators and fill the table. The s_group configuration is static. Orbit can be implemented in failure resistant or failure non-resistant manner. In a non-resistant implementation the Erlang nodes keep only their own partitions of the table; so, when a node fails its part of the table is lost, and Orbit should be restarted. In the failure resistant implementation the partitions are replicated on the neighbouring nodes; thus, when a node fails we may want to restart it in its s_groups.

Mandelbrot Set is a computation oriented benchmark. The benchmark uses escape time algorithm to determine a colour of every pixel on a plot of a given size, i.e. the more iterations it is required for a value to reach an escape point the darker the pixel's colour. The size of the computation depends on the size of the image, the number of iterations to reach an escape condition, and complexity of the escape condition. The s_group configuration is dynamic, the more resources the computation requires the larger number of nodes are involved in the computation. The nodes can be grouped on the basis of locality. When a node fails its computation will be re-spawned to the remaining nodes. If the system requires more resources a new node will be started. Thus, a failure mechanism is not needed.

The exemplars show that applications require different node restarting algorithms if any, i.e. Riak uses information form neighbouring nodes and Mandelbrot set does not require node restarting. In Sim-Diasca and Orbit the nodes can be grouped in s_groups either at launch or dynamically before the programs actually start. In case the s_groups are started at the node launch and the s_groups never change the failed nodes can be restarted using the initial configuration. In case s_groups are started dynamically we have implemented a simple optional node restarting algorithm. The details of the algorithm are presented in Section 2.3.

2.3 Implemented Approach

Applications require different node recovery mechanisms as discussed in Sections 2.1 and 2.2. Therefore, we have implemented a simple mechanism that suits our benchmarks and can be used by setting the `-rsconfig` flag at the launch of Erlang nodes. From (2.1) the approach has the following system state estimation, ξ, parameters:

$$\xi \equiv (Decentralised, Partial, Voluntary, Aperiodic) \tag{2.2}$$

i.e. each node keeps its own s_group information in a configuration file that is updated by the node every time the node s_group configuration changes. The details of the algorithm are as follows. When a node is started and the `-rsconfig` flag is set a new configuration file named `NodeName.config` is created. Every time s_group information is renewed s_group process spawns a process to update the configuration file. The configuration is only updated if the new state information timestamp is greater than the last update. The configuration file is kept in the same directory as the running node.

Consistent with the SD Erlang design principles [CTG+14, Chapter 2], reliable s_groups leave distributed Erlang recovery mechanisms unchanged. In distributed Erlang a node failure can be detected in several ways, e.g. using a supervision behaviour, a heartbeat monitoring by setting the `-heart` flag, or `net_adm:world_list/1,2` and `erlang:monitor_node/2,3` functions. To restart a failed node a script file or `slave:start/3` function can be used. However, when using `slave:start/3` function one should remember that slave nodes automatically terminate when the master process that started the nodes terminates.

Chapter 3

S_groups with the Same Name

It is very difficult to guarantee name uniqueness in a distributed system. We have analysed a number of different approaches, and here is a summary of our analysis.

- Centralised schemes are not scalable and hence are not suitable.
- A flat decentralised scheme either causes a sharp increase of message exchange when an s_group is added or requires non-friendly user names, such as Universally Unique IDentifiers (UUIDs) [LMS05].
- A hierarchical decentralised scheme provides a straightforward solution for name uniqueness but this will require s_groups to be structured in a hierarchical manner which is not suitable for SD Erlang.
- A hybrid scheme also does not seem suitable, i.e. centralised within an s_group and decentralised between s_groups. For example, a leader s_group node keeps information about all s_groups its nodes belong to. Then when a node attempts to create a new s_group only s_group leader nodes exchange messages between each other to check whether an s_group with the same name already exists. However, when the number of s_groups is compatible with the number of nodes the scheme has the same issues as a flat decentralised scheme.
- A leader node approach, i.e. a full s_group name consists of an s_group name and the leader node name, for example, `group1@node1@glasgow.ac.uk`. However, the scheme becomes very complicated when leader nodes together with other s_group nodes fail and we need to pass the s_group leadership and identify s_groups to which the restarted nodes should be added.

Therefore, instead of guaranteeing s_group name uniqueness we guarantee a high probability of name uniqueness and explore node behaviour in cases when s_group names are not unique. For that we modify `s_group:new_s_group` function, i.e. from
`s_group:new_s_group(SGroupName, [Node])` $\rightarrow$ `{SGroupName, [Node]}`
to
`s_group:new_s_group([Node])` $\rightarrow$ `{SGroupName, [Node]}`.
S_group names are generated by `s_group:new_s_group/1` using UUIDs. In SD Erlang we use `uuid` module. In the future this UUID generator may be changed to decrease a probability of generating two identical UUIDs. A discussion on UUID randomness can be found in [IOS10].

In this section we discuss the cases when there are s_groups with the same name and focus on the following two issues.

1. Ensuring that when two nodes from s_groups with the same name connect they behave in a predictable manner.

2. Preventing nodes from joining an s_group when it may cause a confusion, e.g. a node cannot belong to a number of s_groups with the same name.

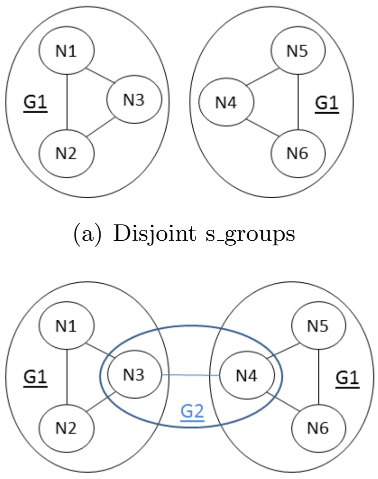
A mathematical description of interconnections between s_groups with the same name is presented below. Here, x, y are nodes; A, A', B are s_groups; and A, A' s_groups have the same s_group name.

- *Disjoint s_groups* contain no nodes which belong to both s_groups that share a name (Figure 3.1(a)), i.e. $\nexists x\ (x \in A, x \in A') \wedge \forall x, y \nexists B\ (x \in A, y \in A', [x, y] \in B)$.

- *Directly overlapping s_groups* contain nodes that belong to multiple s_groups that have the same name (Figure 3.1(b)), i.e. $\exists x\ (x \in A, x \in A')$.

- *Indirectly overlapping s_groups* have no nodes that belong to more than one s_group with a duplicate name but some nodes belong to an s_group with a different name and through it the nodes share a namespace (Figure 3.1(c)), e.g.
$\nexists x\ (x \in A, x \in A') \wedge \exists x, y, B\ (x \in A, y \in A', [x, y] \in B)$.

When we create an s_group or add nodes to an existing s_group we should ensure that no nodes in that s_group belong to different s_groups with the same name. A node may become a member of multiple s_groups that have the same name either when it is started, i.e. at the node launch using s_group configuration, or when nodes are added to existing s_groups, i.e. dynamically using `s_group:new_s_group/2` and `s_group:add_nodes/2` functions. Below we discuss s_group interaction for the case when nodes join s_groups at launch (Section 3.1) and dynamically (Section 3.2).



(a) Disjoint s_groups

(b) Directly overlapping s_groups

(c) Indirectly overlapping s_groups

Figure 3.1: Interconnections between S_groups with the Same Name

3.1 Joining an S_group at Launch

To join an s_group at the node launch the `-config` flag together with a configuration file are used. The configuration file contains information about prospective s_groups. For example, to start a node using `s_group_config.config` file the following command is called:

```
$ erl node1@glasgow.ac.uk -config s_group_config
```

An example of configuration file `s_group_config.config` is presented in Listing 3.1.

Listing 3.1: An example of s_group_config.config file

```
[{kernel, [{s_groups,
  [{group1, normal,
            ['node1@glasgow.ac.uk', 'node2@glasgow.ac.uk',
             'node3@glasgow.ac.uk', 'node4@glasgow.ac.uk']},
   {group2, normal,
            ['node3@glasgow.ac.uk', 'node5@glasgow.ac.uk',
             'node6@glasgow.ac.uk']},
   {group3, normal,
            ['node4@glasgow.ac.uk', 'node7@glasgow.ac.uk',
             'node8@glasgow.ac.uk']}]}]}].
```

Thus, at the node launch we can check the s_group configuration and eliminate undesirable behaviour. The configuration file may contain information either only about own s_groups or about own and remote s_groups. In case the configuration file contains descriptions of multiple s_groups with the same name one of the following actions can be done.

- Merge s_groups that have the same name.
- Consider only the first encounter of the s_group name and ignore the rest.
- Discard s_groups whose names occur more than once in the configuration file.
- Report the error in the configuration file and start the node as a free node.
- Report the error and terminate the node using `init:stop().` function.

From the above alternatives we have implemented the following. In case all s_groups that have the same name are identical we create a single s_group; otherwise we check whether any of these s_groups contain the current node. In case the s_groups that have the same name are remote we keep a record of no s_group with that name. In case the current node is a member of at least one s_group an error is returned and the node terminates.

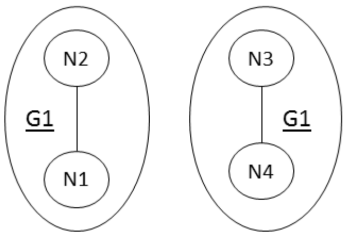
The reason we have decided to terminate the node in case the configuration is wrong is to inform the programmer that the configuration for a particular node is dubious and should be specified to avoid wrong connections and incorrect information sharing. As for the information about the remote nodes, currently, we do not renew s_group information but trust that it is correct, and if the information is ambiguous from the beginning it is better to discard it to eliminate errors.

The cases in which the configuration file contains information only about one s_group that have the same with other s_groups in the system are discussed below. We consider the following types of overlapping: disjoint, direct and indirect overlapping (Chapter 3).

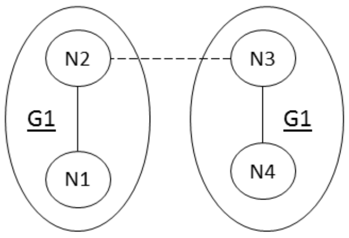
Disjoint S_groups. In case the node has a record of only one s_group with a particular name the node is started and no s_group information is modified. For example, we start four nodes: nodes N1 and N2 are started with configuration from Listing 3.2, and nodes N3 and N4 are started with configuration from Listing 3.3. Nodes from different disjoint s_groups communicate normally and do not share the namespace (Figure 3.2(a)).

Listing 3.2: Configuration 1 for Disjoint S_groups

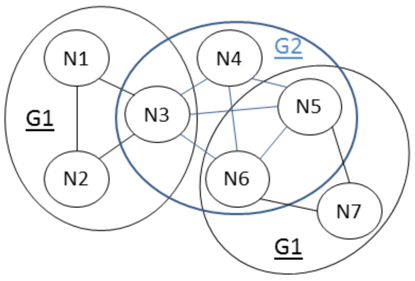
```
[{kernel, [{s_groups,
   [{group1, normal,
         ['node1@glasgow.ac.uk', 'node2@glasgow.ac.uk']}]}]}].
```



(a) Disjoint s_groups



(b) Directly overlapping s_groups



(c) Indirectly overlapping s_groups

Figure 3.2: S_group Interconnections when the S_groups are Joined at Launch

Listing 3.3: Configuration 2 for Disjoint S_groups

```
[{kernel, [{s_groups,
   [{group1, normal,
         ['node3@glasgow.ac.uk', 'node4@glasgow.ac.uk',]}]}]}].
```

Directly Overlapping S_groups may occur when nodes have inconsistent information about s_groups. For example, we start four nodes: nodes N1 and N2 are started with configuration from Listing 3.4, and nodes N3 and N4 are started with configuration from Listing 3.5. As nodes N2 and N3 have different information about group1 the nodes do not synchronise and have a synchronisation error, i.e. nodes N1 and N2 belong to one s_group and nodes N3 and N4 belong to another s_group that have the same name group1. The s_groups do not share connections and namespace (Figure 3.2(b)).

Listing 3.4: Configuration 1 for Directly Overlapping S_groups

```
[{kernel, [{s_groups,
   [{group1, normal,
             ['node1@glasgow.ac.uk', 'node2@glasgow.ac.uk',
              'node3@glasgow.ac.uk']}]}]}]}].
```

Listing 3.5: Configuration 2 for Directly Overlapping S_groups

```
[{kernel, [{s_groups,
   [{group1, normal,
              ['node2@glasgow.ac.uk', 'node3@glasgow.ac.uk',
               'node4@glasgow.ac.uk',]}]}]}].
```

Indirectly Overlapping S_groups may occur when nodes have partial information about remote s_groups. For example, we start seven nodes: nodes N1, N2, N3 are started using configuration from Listing 3.6, and nodes N4, N5, N6, N7 are started using configuration from Listing 3.7. Independently of the order we start the nodes they do not share connections and name spaces (Figure 3.2(c)).

Listing 3.6: Configuration 1 for Indirectly Overlapping S_groups

```
[{kernel, [{s_groups,
   [{group1, normal,
        ['node1@glasgow.ac.uk', 'node2@glasgow.ac.uk',
         'node3@glasgow.ac.uk']},
    {group2, normal,
        ['node3@glasgow.ac.uk', 'node4@glasgow.ac.uk',
         'node5@glasgow.ac.uk', 'node6@glasgow.ac.uk']}]}]}].
```

Listing 3.7: Configuration 2 for Indirectly Overlapping S_groups

```
[{kernel, [{s_groups,
   [{group1, normal,
        ['node5@glasgow.ac.uk', 'node6@glasgow.ac.uk',
         'node7@glasgow.ac.uk',]},
    {group2, normal,
        ['node3@glasgow.ac.uk', 'node4@glasgow.ac.uk',
         'node5@glasgow.ac.uk', 'node6@glasgow.ac.uk']}]}]}].
```

3.2 Joining an S_group Dynamically

A node can join an s_group dynamically using `s_group:new_s_group/1` or `s_group:add_nodes/2` functions defined in Sections 3, 4.2.1, and in [CLTG14]. In both cases we distinguish two types of nodes: a node that initiates the action and nodes that join the s_group. Additionally, `s_group:add_nodes/2` function has the third type of nodes, i.e. nodes which are already members of the s_group.

Disjoint S_groups may occur when s_groups with the same name are started independently from each other. When nodes from disjoint s_groups connect they share neither connections nor namespaces.

Directly Overlapping S_groups. To prevent directly overlapping s_groups the initiating node first checks whether the new s_group's name is identical to an s_group the node already belongs to. If it is then a new s_group name is generated and the checking is repeated. When a node receives a request to join an s_group it also first checks whether it already belongs to an s_group with the same name. Only nodes that return a negative response are added to the s_group.

Indirectly Overlapping S_groups may occur as a result of using `s_group:new_s_group/1` or `s_group:add_nodes/2` functions. When nodes from indirectly overlapping s_groups connect they do not share connections and namespaces.

3.2.1 Summary.

In this section we have discussed dynamic and at launch s_group creation, and measures we have taken in SD Erlang to minimise the probability of presence of s_groups that have the same name in the system (Chapters 3 and 3.1). For each type of s_group creation we have discussed three ways of interconnections between s_groups that have the same name, i.e. disjoint s_groups, directly overlapping s_groups, and indirectly overlapping s_groups (Sections 3.1 and 3.2). Here, have covered the cases when the interconnections may occur, and how nodes and the system behave.

We recommend using at launch state configuration only in cases when nodes need to know information about other s_groups in the system from the start, and s_group configuration is not likely to change. It is up to the programmer to ensure that at launch s_group configuration is correct, and no s_groups have identical s_group names, e.g. starting the whole system with a single configuration file to avoid copy-paste errors. Otherwise, dynamic s_group creation provides a higher probability of unique s_group names. To avoid s_group name clash we also advise using either only static or only dynamic s_group creation.

Chapter 4

S_group Operational Semantics

As requested at the 1st year review we present a small-step transition semantics for the s_group operations provided by SD Erlang. For simplicity the current semantics ignores failures. We start by defining an abstract state of SD Erlang systems (Section 4.1), and then define each operation as a transition between states (Section 4.2).

4.1 SD Erlang State

The state of an SD Erlang system, and associated abstract syntax variables, are defined in Figure 4.1. The state of a system is modelled as a four tuple comprising a set of *s_group*s, a set of *free_group*s, a set of *free_hidden_group*s, and a set of *node*s. Each type of groups is associated with nodes and has a namespace. An *s_group* additionally has a name, whereas a *free_hidden_group* consists of only one node, i.e. a hidden node simultaneously acts as a node and as a group, because as a group it has a namespace but does not share it with any other node. Free normal and hidden groups have no names, and are uniquely defined by the nodes associated with them. Therefore, group names, *gr_names*, are either *NoGroup* or a set of *s_group_name*s. A *namespace* is a set of *name* and process id, *pid*, pairs and is replicated on all nodes of the associated group.

A *node* has the following parameters: *node_id* identifier, *node_type* that can be either hidden or normal, *connections*, and *group_names*, i.e. names of groups the node belongs to. The node can belong to either a list of s_groups or one of the free groups. The type of the free group is defined by the node type. Connections are a set of *node_id*s.

SD Erlang State Property. Every node in an SD Erlang state is a member of one of the three classes of groups: *s_group*, *free_group*, or

$$
\begin{aligned}
(grs, fgs, fhs, nds) \in \{state\} &\equiv \{(\{s\_group\}, \{free\_group\}, \{free\_hidden\_group\}, \{node\})\} \\
gr \in grs \equiv \{s\_group\} &\equiv \{(s\_group\_name, \{node\_id\}, namespace)\} \\
fg \in fgs \equiv \{free\_group\} &\equiv \{(\{node\_id\}, namespace)\} \\
fh \in fhs \equiv \{free\_hidden\_group\} &\equiv \{(node\_id, namespace)\} \\
nd \in nds \equiv \{node\} &\equiv \{(node\_id, node\_type, connections, gr\_names)\} \\
gs \in \{gr\_names\} &\equiv \{NoGroup, \{s\_group\_name\}\} \\
ns \in \{namespace\} &\equiv \{\{(name, pid)\}\} \\
cs \in \{connections\} &\equiv \{\{node\_id\}\} \\
nt \in \{node\_type\} &\equiv \{Normal, Hidden\} \\
s &\in \{NoGroup, s\_group\_name\} \\
n &\in \{name\} \\
p &\in \{pid\} \\
ni &\in \{node\_id\} \\
nis &\in \{\{node\_id\}\} \\
m &\in \{message\}
\end{aligned}
$$

Figure 4.1: SD Erlang State

$free\_hidden\_group$. The three classes of groups partition the set of nodes. That is, for any state (grs, fgs, fhs, nds) $\{\Pi_{node\_id} grs, \Pi_{node\_id} fgs, \Pi_{node\_id} fhs\}$ is a partition of $\Pi_{node\_id} nds$ where Π_{node_id} is projection onto the *node_id* attribute, or set of attributes, of the tuples.

Assumptions. We make the following two assumptions to simplify the state transitions.

1. No two s_groups have the same name, that is all *s_group_name*s are unique. We discuss s_group name uniqueness in Chapter 3.
2. All *node_id*s identify some node. More formally, for all *node_id*s occurring in some state (grs, fgs, fhs, nds), there exists some node in *nds* with that *node_id*.

As we continue the work on the SD Erlang semantics we plan to relax these assumptions.

4.2 Transitions

The transitions we present in Section 4.2.1 have the following form:

$$(state, \text{command}, ni) \longrightarrow (state', value)$$

meaning that executing command on node ni in $state$ returns $value$ and transitions to $state'$. The transitions use a number of auxiliary functions defined in Section 4.2.2.

In the following $\oplus$ denotes disjoint set union. By $xxs \equiv \bigoplus\{xx \mid ...\}$ we mean that elements from all generated xx sets are accumulated in one xxs set.

4.2.1 SD Erlang Functions

In this section we present the transitions of fifteen SD Erlang functions. The following nine functions change their state after the transition: register_name/3, re_register_name/3, unregister_name/2, whereis_name/3, send/2, new_s_group/2, delete_s_group/1, add_nodes/2, remove_nodes/2. Whereas the other six functions only return some state information but do not change the state after the transition: send/3, whereas_name/2, registered_names/1, own_nodes/0, own_nodes/1, own_s_groups/0.

The implementation and description of the functions are discussed in [CLTG14].

register_name. When registering name n for pid p in s_group s the pair (n, p) is added to the namespace ns of the s_group only if neither n nor p appears in the namespace, and node ni is a member of s_group s.

$$
\begin{aligned}
&((grs, fgs, fhs, nds), \text{register\_name}(s, n, p), ni)\\
&\quad\longrightarrow ((\{(s, \{ni\} \oplus nis, \{(n, p)\} \oplus ns)\} \oplus grs', fgs, fhs, nds), \text{True})\\
&\qquad\quad \text{If } (n, \_) \notin ns \wedge (\_, p) \notin ns\\
&\quad\longrightarrow ((grs, fgs, fhs, nds), \text{False})\\
&\qquad\quad \text{Otherwise}\\
&\text{where}\\
&\qquad\quad \{(s, \{ni\} \oplus nis, ns)\} \oplus grs' \equiv grs
\end{aligned}
$$

re_register_name. When re_registering name n for different pid p in s_group s the name is re-registered only if the pid is not already registered in namespace , and node ni is a member of s_group s.

$$
\begin{array}{lr}
((grs, fgs, fhs, nds), \text{re\_register\_name}(s, n, p), ni) & \\
\quad \longrightarrow ((\{(s, \{ni\} \oplus nis, ns' \oplus \{(n, p)\})\} \oplus grs', fgs, fhs, nds), \text{True}) & \text{If } (\_, p) \notin ns \\
\quad \longrightarrow ((grs, fgs, fhs, nds), \text{False}) & \text{Otherwise}
\end{array}
$$

where

$$\{(s, \{ni\} \oplus nis, ns)\} \oplus grs' \equiv grs$$
$$ns' \equiv ns - \{(n, \_)\}$$

unregister_name. To unregister name n from namespace ns of s_group s node ni should belong to s_group s.

$$
\begin{array}{lr}
((grs, fgs, fhs, nds), \text{unregister\_name}(s, n), ni) & \\
\quad \longrightarrow ((\{(s, \{ni\} \oplus nis, ns - \{(n, p)\})\} \oplus grs', fgs, fhs, nds), \text{True}) & \text{If } (n, \_) \in ns \\
\quad \longrightarrow ((grs, fgs, fhs, nds), \text{True}) & \text{Otherwise}
\end{array}
$$

where

$$\{(s, \{ni\} \oplus nis, ns)\} \oplus grs' \equiv grs$$

registered_names function returns a list of names registered in s_group s if node ni belongs the s_group, empty list otherwise. IsSGroupSNode/3 and OutputNs/2 are auxiliary functions defined in Section 4.2.2.

$$
\begin{array}{lr}
((grs, fgs, fhs, nds), \text{registered\_names}(s), ni) & \\
\quad \longrightarrow ((grs, fgs, fhs, nds), nss) & \text{If IsSGroupSNode}(ni, s, grs) \\
\quad \longrightarrow ((grs, fgs, fhs, nds), \{\}) & \text{Otherwise}
\end{array}
$$

where

$$\{(s, \{ni\} \oplus nis, ns)\} \oplus grs' \equiv grs$$
$$nss \equiv \text{OutputNs}(s, ns)$$

whereis_name/2 function returns pid p registered as name n in s_group s if node ni belongs to the s_group, `undefined` otherwise.

$$
\begin{array}{lr}
((grs, fgs, fhs, nds), \text{whereis\_name}(s, n), ni) & \\
\quad \longrightarrow ((grs, fgs, fhs, nds), p) & \text{If IsSGroupSNode}(ni, s, grs) \\
\quad \longrightarrow ((grs, fgs, fhs, nds), \text{undefined}) & \text{Otherwise}
\end{array}
$$

where

$$\{(s, \{ni\} \oplus nis, ns)\} \oplus grs' \equiv grs$$
$$p \equiv \text{FindName}(ni, s, n, grs, fgs, fhs, nds)$$

whereis_name/3. Depending on the type of nodes ni and ni' a search of name n from group s on a remote node ni' may lead to establishing new connections between nodes ni, ni', and the node on which process p actually resides.

$$
\begin{aligned}
&((grs, fgs, fhs, nds), \text{whereis\_name}(ni', s, n), ni)\\
&\quad\longrightarrow ((grs, fgs', fhs, nds'), p)\\
&\text{where}\\
&\qquad (fgs', nds') \equiv \text{AddConnections}(ni, ni', grs, fgs, fhs, nds)\\
&\qquad p \equiv \text{FindName}(ni', s, n, grs, fgs', fhs, nds')
\end{aligned}
$$

send. In SD Erlang we have implemented two `send` functions, i.e. `send/3` and `send/4`. Function `send/3` is defined in Erlang as a `where_is/2` to locate pid p of process name n from s_group s, followed by `send/2` to send message m to the pid (Listing 4.1).

Listing 4.1: send/3 Function in Erlang-like Syntax

```
send(s,n,m) ->
    p = whereis_name(s,n),
    send(p,m).
```

Function `send/4` is defined in Erlang as a `where_is/3` to locate pid p of process name n from s_group s and node ni', followed by `send/2` to send message m to the pid (Listing 4.2).

Listing 4.2: send/4 Function in Erlang-like Syntax

```
send(ni',s,n,m) ->
     p = whereis_name(ni',s,n),
     send(p,m).
```

Transitions of `whereis_name(ni',s,n)` and `whereis_name(s,n)` functions are presented above, and a transition of `send(p,m)` function is as follows:

$$
\begin{aligned}
&((grs, fgs, fhs, nds), \text{send}(p, m), ni)\\
&\quad\longrightarrow ((grs, fgs', fhs, nds'), p)\\
&\text{where}\\
&\qquad ni' \equiv \text{node}(p)\\
&\qquad (fgs', nds') \equiv \text{AddConnections}(ni, ni', grs, fgs, fhs, nds)
\end{aligned}
$$

"node(p)" is a primitive distributed Erlang function `node(Pid)→Node` that identifies the *node_id* of the node hosting process p. The passing of the message is not modelled in our state.

new_s_group. When we create a new s_group s, the s_group together with its nodes nis are added to the list of s_groups. If before joining the s_group nodes nis free then the nodes are removed from free groups. The new s_group has an empty namespace.

$$
\begin{array}{ll}
((grs, fgs, fhs, nds), \text{new\_s\_group}(s, nis), ni) & \\
\quad \longrightarrow ((grs', fgs', fhs', nds''), (s, nis)) & \text{If } ni \in nis \\
\quad \longrightarrow ((grs, fgs, fhs, nds), \text{Error}) & \text{Otherwise} \\
\text{where} & \\
\qquad nds' \equiv \text{InterConnectNodes}(nis, nds) & \\
\qquad nds'' \equiv \text{AddSGroup}(s, nis, nds') & \\
\qquad grs' \equiv grs \oplus \{(s, nis, \{\})\} & \\
\qquad (fgs', fhs') \equiv \text{RemoveNodes}(nis, fgs, fhs) &
\end{array}
$$

add_nodes. For node ni to add nodes identified by nis to s_group s node ni should belong to that s_group. S_group s membership is added to nds' nodes identified by nis node ids, and the existing and the new s_group nodes interconnect. The nodes that were free before joining the s_group are removed from the corresponding free groups fgs' and fhs'.

$$
\begin{array}{l}
((grs, fgs, fhs, nds), \text{add\_nodes}(s, nis), ni) \\
\quad \longrightarrow ((grs'', fgs', fhs', nds''), (s, nis)) \\
\qquad \text{If IsSGroupSNode}(ni, s, grs) \\
\quad \longrightarrow ((grs, fgs, fhs, nds), \text{Error}) \\
\qquad \text{Otherwise} \\
\text{where} \\
\qquad \{(s, \{ni\} \oplus nis', ns)\} \oplus grs' \equiv grs \\
\qquad nds' \equiv \text{InterConnectNodes}(\{ni\} \oplus nis \oplus nis', nds) \\
\qquad nds'' \equiv \text{AddSGroup}(s, nis, nds') \\
\qquad grs'' \equiv \{(s, (\{ni\} \oplus nis) \oplus nis', ns)\} \oplus grs' \\
\qquad (fgs', fhs') \equiv \text{RemoveNodes}(nis, fgs, fhs)
\end{array}
$$

delete_s_group. To delete s_group s node defined by ni node id should be a member of that s_group. When s_group s is deleted its membership is removed from group names of nds identified by ni and nis. In case the nodes become free, we add new free nodes to the corresponding free groups.

$$\begin{aligned}
&((\mathit{grs}, \mathit{fgs}, \mathit{fhs}, \mathit{nds}), \text{delete\_s\_group}(s), \mathit{ni}) \\
&\quad \longrightarrow ((\mathit{grs}', \mathit{fgs}', \mathit{fhs}', \mathit{nds}''), \text{True}) && \text{If IsSGroupSNode}(\mathit{ni}, s, \mathit{grs}) \\
&\quad \longrightarrow ((\mathit{grs}, \mathit{fgs}, \mathit{fhs}, \mathit{nds}), \text{False}) && \text{Otherwise} \\
&\text{where} \\
&\qquad \{(s, \{\mathit{ni}\} \oplus \mathit{nis}, \mathit{ns})\} \oplus \mathit{grs}' \equiv \mathit{grs} \\
&\qquad \mathit{nds}' \equiv \text{RemoveSGroup}(s, \{\mathit{ni}\} \oplus \mathit{nis}, \mathit{nds}) \\
&\qquad (\mathit{fgs}', \mathit{fhs}', \mathit{nds}'') \equiv \text{NewFreeNodes}(\mathit{fgs}, \mathit{fhs}, \mathit{nds}')
\end{aligned}$$

remove_nodes. To remove nodes identified by *nis* node ids from s_group *s* the nodes should be members of that s_group. Additionally, *ni* node should be also a member of s_group *s* and it cannot remove itself from an s_group. The s_group *s* membership is removed from group names of *nds* identified by *nis*. In case the nodes become free, we add new free nodes to the corresponding free groups.

$$\begin{aligned}
&((\mathit{grs}, \mathit{fgs}, \mathit{fhs}, \mathit{nds}), \text{remove\_nodes}(s, \mathit{nis}), \mathit{ni}) \\
&\quad \longrightarrow ((\mathit{grs}'', \mathit{fgs}', \mathit{fhs}', \mathit{nds}''), \text{True}) \\
&\qquad \text{If } \mathit{ni} \notin \mathit{nis} \land \text{IsSGroupSNode}(\mathit{ni}, s, \mathit{grs}) \\
&\quad \longrightarrow ((\mathit{grs}, \mathit{fgs}, \mathit{fhs}, \mathit{nds}), \text{False}) \\
&\qquad \text{Otherwise} \\
&\text{where} \\
&\qquad \{(s, (\{\mathit{ni}\} \oplus \mathit{nis}) \oplus \mathit{nis}', \mathit{ns})\} \oplus \mathit{grs}' \equiv \mathit{grs} \\
&\qquad \mathit{grs}'' \equiv \{(s, \{\mathit{ni}\} \oplus \mathit{nis}', \mathit{ns})\} \oplus \mathit{grs}' \\
&\qquad \mathit{nds}' \equiv \text{RemoveSGroup}(s, \mathit{nis}, \mathit{nds}) \\
&\qquad (\mathit{fgs}', \mathit{fhs}', \mathit{nds}'') \equiv \text{NewFreeNodes}(\mathit{fgs}, \mathit{fhs}, \mathit{nds}')
\end{aligned}$$

own_nodes function has two versions, i.e. `own_nodes/0` and `own_nodes/1`. When no parameter is specified, i.e. `own_nodes()`, the function returns a list of nis' node ids of the nodes that belong to the same groups as node *ni*.

$$\begin{aligned}
&((\mathit{grs}, \mathit{fgs}, \mathit{fhs}, \mathit{nds}), \text{own\_nodes}(), \mathit{ni}) \\
&\quad \longrightarrow ((\mathit{grs}, \mathit{fgs}, \mathit{fhs}, \mathit{nds}), \mathit{nis}') \\
&\text{where} \\
&\qquad \mathit{nis}' \equiv \text{OwnGroupNis}(\mathit{ni}, \mathit{grs}, \mathit{fgs}, \mathit{fhs})
\end{aligned}$$

When s_group *s* is specified, i.e. `own_nodes(s)` the function returns a list of nis' node ids of the nodes that belong to the s_group. In case node

ni does not belong to s_group s, an empty list is returned.

$$
\begin{array}{ll}
((grs, fgs, fhs, nds), \text{own\_nodes}(s), ni) & \\
\quad \longrightarrow ((grs, fgs, fhs, nds), nis') & \text{If IsSGroupSNode}(ni, s, grs) \\
\quad \longrightarrow ((grs, fgs, fhs, nds), \{\}) & \text{Otherwise} \\
\text{where} & \\
\qquad \{(s, \{ni\} \oplus nis, ns)\} \oplus grs' \equiv grs & \\
\qquad nis' \equiv \{ni\} \oplus nis &
\end{array}
$$

own_s_groups function returns a list of s_groups and their nodes to which ni node belongs to if node ni is an s_group node, empty list otherwise.

$$
\begin{array}{ll}
((grs, fgs, fhs, nds), \text{own\_s\_groups}(), ni) & \\
\quad \longrightarrow ((grs, fgs, fhs, nds), snis) & \text{If IsSGroupNode}(ni, grs) \\
\quad \longrightarrow ((grs, fgs, fhs, nds), \{\}) & \text{Otherwise} \\
\text{where} & \\
\qquad snis \equiv \text{OutputOwnSGroups}(ni, grs) &
\end{array}
$$

4.2.2 Auxiliary Functions

In this section we present auxiliary functions introduced in Section 4.2.1.

IsSGroupNode function returns True if node ni is a member of some s_group s, False otherwise.

$$\text{IsSGroupNode}(ni, grs) = \exists s, nis, ns, grs' \ . \ \{(s, \{ni\} \oplus nis, ns)\} \oplus grs' \equiv grs$$

IsFreeNormalNode function returns True if node ni is a member of some free normal group, False otherwise.

$$\text{IsFreeNormalNode}(ni, fgs) = \exists nis, ns, fgs' \ . \ \{(\{ni\} \oplus nis, ns)\} \oplus fgs' \equiv fgs$$

IsFreeHiddenNode function returns True if node ni is a member of some free hidden group, False otherwise.

$$\text{IsFreeHiddenNode}(ni, fhs) = \exists ns, fhs' \ . \ \{(ni, ns)\} \oplus fhs' \equiv fhs$$

SameFreeGroup function returns True if nodes ni and ni' are members of the same free normal group, False otherwise.

$$\text{SameFreeGroup}(ni, ni', fgs) = \exists nis, ns, fgs' \ . \ \{(\{ni, ni'\} \oplus nis, ns)\} \oplus fgs' \equiv fgs$$

SameSGroup function returns True if nodes ni and ni' are members of the same s_group, False otherwise.

$$\text{SameSGroup}(ni, ni', grs) = \exists s, nis, ns, grs' \ . \ \{(s, \{ni, ni'\} \oplus nis, ns)\} \oplus grs' \equiv grs$$

AddConnections function systematically checks types of nodes ni and ni', and then adds corresponding connections. Here, a connection between the nodes may already exist. The first outcome (fgs, nds) occurs when the nodes are in the same free normal group or the same s_group and hence are connected. The second outcome (fgs, nds') occurs in one of the following cases: 1) node ni is an s_group node, *and* nodes ni and ni' are not in the same s_group; note that ni' may be an s_group node, a free normal node, or a free hidden node; 2) node ni is a free hidden node; 3) node ni is a free normal node *and* node ni' is either an s_group node or a free hidden node. The third outcome (fgs'', nds'') occurs when nodes ni and ni' are in different free normal groups.

$$\begin{aligned}
&\text{AddConnections}(ni, ni', grs, fgs, fhs, nds) \\
&= (fgs, nds) && \text{If SameFreeGroup}(ni, ni', fgs) \\
& && \lor \text{ SameSGroup}(ni, ni', grs) \\
&= (fgs, nds') && \text{If (IsSGroupNode}(ni, grs) \\
& && \land\neg \text{ SameSGroup}(ni, ni', grs)) \\
& && \lor \text{ IsFreeHiddenNode}(ni, fhs) \\
& && \lor \text{ (IsFreeNormalNode}(ni, fgs) \\
& && \land\neg \text{ IsFreeNormalNode}(ni', fgs)) \\
&= (fgs'', nds'') && \text{If } ni \in nis1 \land ni' \in nis2 \\
&= (fgs, nds) && \text{Otherwise}
\end{aligned}$$

where

$$\begin{aligned}
& nds' \equiv \text{InterConnectNodes}(\{ni, ni'\}, nds) \\
& (\{(nis1, ns1)\} \oplus \{(nis2, ns2)\}) \oplus fgs' \equiv fgs \\
& fgs'' \equiv \{(nis1 \oplus nis2, ns1 \oplus ns2)\} \oplus fgs' \\
& nds'' \equiv \text{InterConnectNodes}(nis1 \oplus nis2, nds)
\end{aligned}$$

InterConnectNodes function interconnects nodes from nds identified by nis node ids.

$$\begin{aligned}
&\text{InterConnectNodes}(nis, nds) \\
&= nds \ \cup \ \{(ni, nt, (cs \oplus nis) - \{ni\}, gs) \mid (ni, nt, cs, gs) \in nds, ni \in nis\}
\end{aligned}$$

AddSGroup function adds membership of s_group s to all nodes identified by nis node ids.

$$
\begin{aligned}
&\text{AddSGroup}(s, nis, nds) = (nds - nds') \oplus nds'' \\
&\quad \text{where} \\
&\qquad nds' \equiv \{(ni, nt, cs, gs) \mid (ni, nt, cs, gs) \in nds, ni \in nis\} \\
&\qquad nds'' \equiv \{(ni, nt, cs, \text{AddSGroupS}(s, gs)) \mid (ni, nt, cs, gs) \in nds'\}
\end{aligned}
$$

AddSGroupS function adds s_group s to a list of s_groups.

$$
\begin{aligned}
&\text{AddSGroupS}(s, gs) \\
&\qquad = \{s\} && \text{If } gs \equiv NoGroup \\
&\qquad = gs \oplus \{s\} && \text{Otherwise}
\end{aligned}
$$

RemoveSGroup function removes membership of s_group s from all nodes identified by nis node ids.

$$
\begin{aligned}
&\text{RemoveSGroup}(s, nis, nds) = (nds - nds') \oplus nds'' \\
&\quad \text{where} \\
&\qquad nds' \equiv \{(ni, nt, cs, gs) \mid (ni, nt, cs, gs) \in nds, ni \in nis\} \\
&\qquad nds'' \equiv \{(ni, nt, cs, gs) \mid (ni, nt, cs, gs \oplus \{s\}) \in nds'\}
\end{aligned}
$$

RemoveNodes function removes node ids identified by nis from free normal groups fgs and free hidden groups fhs.

$$
\begin{aligned}
&\text{RemoveNodes}(nis, fgs, fhs) = (fgs'', fhs') \\
&\quad \text{where} \\
&\qquad fgs' \equiv \{(\{ni\} \oplus nis', ns') \mid (\{ni\} \oplus nis', ns') \in fgs, ni \in nis\} \\
&\qquad fgs'' \equiv (fgs - fgs') \oplus \{(nis', ns') \mid (\{ni\} \oplus nis', ns') \in fgs', ni \in nis\} \\
&\qquad fhs' \equiv fhs - \{(ni, ns) \mid (ni, ns) \in fhs, ni \in nis\}
\end{aligned}
$$

FindName function identifies pid p of name n registered in group s. Here, s can be $NoGroup$ or an s_group name.

$$\begin{aligned}
&\text{FindName}(ni, s, n, grs, fgs, fhs)\\
&\qquad = p1 && \text{If } s \equiv NoGroup \wedge nt \equiv Normal\\
&\qquad = p2 && \text{If } s \equiv NoGroup \wedge nt \equiv Hidden\\
&\qquad = p3 && \text{Otherwise}\\
&\quad \text{where}\\
&\qquad\quad \{(ni, nt, cs, gs)\} \oplus nds' \equiv nds\\
&\qquad\quad \{(\{ni\} \oplus nis, \{(n, p1)\} \oplus ns1)\} \oplus fgs' \equiv fgs\\
&\qquad\quad \{(ni, \{(n, p2)\} \oplus ns2)\} \oplus fhs' \equiv fhs\\
&\qquad\quad \{(s, \{ni\} \oplus nis, \{(n, p3)\} \oplus ns3)\} \oplus grs' \equiv grs
\end{aligned}$$

NewFreeNodes function places new free nodes to corresponding free groups. The groups are determined by the node types and connections.

$$\begin{aligned}
&\text{NewFreeNodes}(fgs, fhs, nds) = (fgs', fhs', nds'')\\
&\qquad \text{where}\\
&\qquad\quad nds' \equiv \{(ni, nt, cs, NoGroup) \mid (ni, nt, cs, \{\}) \in nds\}\\
&\qquad\quad fhs' \equiv \text{AddFreeHiddenGroup}(fhs, nds')\\
&\qquad\quad (fgs', nds'') \equiv \text{AddFreeNormalGroup}(fgs, nds, nds')
\end{aligned}$$

AddFreeHiddenGroup function adds new free hidden groups to fhs.

$$\text{AddFreeHiddenGroup}(fhs, nds) = fhs \oplus \{(ni, \{\}) \mid (ni, Hidden, cs, NoGroup) \in nds\}$$

AddFreeNormalGroup function distributes new free normal nodes defined by nis from nds' to corresponding free normal groups. The groups are defined by the node connections css' to existing free nodes. In case a new node has no connections with existing free normal nodes a new free normal group is created.

$$\begin{aligned}
&\text{AddFreeNormalGroup}(fgs, nds, nds') = (fgs', nds'')\\
&\qquad \text{where}\\
&\qquad\quad nis \equiv \{ni \mid (ni, Normal, cs, NoGroup) \in nds'\}\\
&\qquad\quad css \equiv \bigoplus \{cs \mid (ni, Normal, cs, NoGroup) \in nds'\}\\
&\qquad\quad css' \equiv \{ni \mid (nis' \oplus \{ni\}, ns) \in fgs, ni \in css\}\\
&\qquad\quad (fgs', nds'') \equiv \text{MergeFreeNormalGroups}(nis, css', fgs, nds)
\end{aligned}$$

MergeFreeNormalGroups function merges free normal groups which nodes are connected with each other. The function also adds missing connections between nodes from the same free normal group.

$$\begin{aligned}
&\text{MergeFreeNormalGroups}(\mathit{nis}, \mathit{css}, \mathit{fgs}, \mathit{nds}) \\
&\qquad = (\mathit{fgs}, \mathit{nds}) && \text{If } \mathit{nis} \equiv \{\} \\
&\qquad = (\mathit{fgs}', \mathit{nds}) && \text{If } \mathit{nis} \not\equiv \{\} \wedge \mathit{css} \equiv \{\} \\
&\qquad = (\mathit{fgs}'', \mathit{nds}') && \text{Otherwise} \\
&\quad \text{where} \\
&\qquad \mathit{fgs}' \equiv \mathit{fgs} \oplus \{(\mathit{nis}, \{\})\} \\
&\qquad \mathit{fgs\_r} \equiv \text{FindFreeNormalGroups}(\mathit{css}, \mathit{fgs}) \\
&\qquad \mathit{nis1} \equiv \bigoplus \{\mathit{nis}' \mid (\mathit{nis}', \mathit{ns}) \in \mathit{fgs\_r}\} \\
&\qquad \mathit{nis}'' \equiv \mathit{nis1} \oplus \mathit{nis} \\
&\qquad \mathit{ns}' \equiv \bigoplus \{\mathit{ns} \mid (\mathit{nis}', \mathit{ns}) \in \mathit{fgs\_r}\} \\
&\qquad \mathit{fgs}'' \equiv (\mathit{fgs} - \mathit{fgs\_r}) \oplus \{(\mathit{nis}'', \mathit{ns}')\} \\
&\qquad \mathit{nds}' \equiv \text{InterConnectNodes}(\mathit{nis}'', \mathit{nds})
\end{aligned}$$

FindFreeNormalGroups function returns a list of free normal groups that contain nodes identified by nis node ids.

$$\text{FindFreeNormalGroups}(\mathit{nis}, \mathit{fgs}) = \{(\mathit{nis}' \oplus \{\mathit{ni}\}, \mathit{ns}) \mid (\mathit{nis}' \oplus \{\mathit{ni}\}, \mathit{ns}) \in \mathit{fgs}, \mathit{ni} \in \mathit{nis}\}$$

OwnGroupNis function returns a list of nis' node ids of the nodes that belong to the same groups as ni node.

$$\begin{aligned}
&\text{OwnGroupNis}(\mathit{ni}, \mathit{grs}, \mathit{fgs}, \mathit{fhs}) \\
&\qquad = \{\mathit{ni}\} \oplus \mathit{nis} && \text{If IsFreeNormalNode}(\mathit{ni}, \mathit{fgs}) \\
&\qquad = \{\mathit{ni}\} && \text{If IsFreeHiddenNode}(\mathit{ni}, \mathit{fhs}) \\
&\qquad = \mathit{nis}' && \text{Otherwise} \\
&\quad \text{where} \\
&\qquad \{(\{\mathit{ni}\} \oplus \mathit{nis}, \mathit{ns})\} \oplus \mathit{fgs}' \equiv \mathit{fgs} \\
&\qquad \mathit{nis}' \equiv \text{OwnSGroupNis}(\mathit{ni}, \mathit{grs})
\end{aligned}$$

OwnSGroupNis function returns a list of nis' node ids of the nodes that belong to the same s_groups as ni node.

$$\begin{aligned}
&\text{OwnSGroupNis}(\mathit{ni}, \mathit{grs}) = \mathit{nis}' \oplus \{\mathit{ni}\} \\
&\quad \text{where} \\
&\qquad \mathit{nis}' \equiv \bigoplus \{\mathit{nis} \mid (s, \{\mathit{ni}\} \oplus \mathit{nis}, \mathit{ns}) \in \mathit{grs}\}
\end{aligned}$$

OutputNs function returns a list of process names registered in *ns* namespace of s_group *s*.

$$\text{OutputNs}(s, ns) = \{(s, n) \mid (n, p) \in ns\}$$

OutputOwnSGroups function returns information about s_groups to which *ni* node belongs to, i.e. a list of s_group names *s* and node ids of nodes from those s_groups.

$$\text{OutputOwnSGroups}(ni, grs) = \{(s, \{ni\} \oplus nis) \mid (s, \{ni\} \oplus nis, ns) \in grs\}$$

Chapter 5

Validation of S_group Semantics and Implementation

In addition to defining the operational semantics of SD Erlang, we have also validated the consistency between the formal semantics and the SD Erlang implementation. Our approach is based on *property-based testing*, in particular we use the Erlang QuickCheck [AHJW06] tool developed by Quivq. The testing revealed s_group bugs in both the semantics and the s_group implementation; it gives us high confidence on the consistency between the formal semantics of s_groups and the implementation. Although SD Erlang is the subject under test in this case, the testing approach is applicable to the validation of formal operational semantics in general. While Quivq QuickCheck is the tool we use for testing, another property-based testing tool *PropEr* [PAS11] written in Erlang and inspired by QuickCheck could be an alternative.

In this section we provide a brief introduction of property-based testing and QuickCheck's `eqc_statem` in Section 5.1, and an overview of the testing approach in Section 5.2.

5.1 Property-based Testing and QuickCheck

Property-Based Testing (PBT) provides a powerful and high-level approach to testing. In PBT system behaviour is specified by properties expressed in a logical form rather than focusing on individual test cases. For example, a function without side effects may be specified by means of the full input/output relation using a universal quantification over all the inputs; a stateful system will be described by means of a model, which is an extended finite state machine. The system is then tested by checking whether it has the required properties for randomly generated data; the randomly generated data may be inputs to functions, sequences of API calls to the stateful system, or other representations of test cases.

QuickCheck supports random testing of Erlang programs. Properties of the programs are stated in a subset of first-order logic embedded in Erlang syntax. QuickCheck verifies these properties for collections of randomly generated Erlang values with user guidance in defining the generators where necessary. When a counterexample is found, QuickCheck tries to generate a simpler – and thus more comprehensible – counterexample, in a constructive manner; this process is called *shrinking*. *PropEr* testing tool has a very similar user interface.

QuickCheck is the tool of choice for the testing of SD Erlang developed by QuivQ. In particular, we utilise its support for extended finite state machines through the `eqc_statem` behaviour. As illustrated in Figure 5.1, with `eqc_statem` the user defines an abstract model of the System Under Test (SUT), including an abstraction of the state of the SUT itself. The model includes an initial abstract state in which test cases begin and describes how each command changes the state. The state is used by QuickCheck both during test case generation and during test execution. For each command a number of properties can be described, e.g.

- *preconditions* to decide whether or not to include a candidate command in test cases;
- *postconditions* to check whether the value returned by the executed command is correct;
- a description of the changes in the abstract state as a result of command execution;
- a method to generate an appropriate function call to appear next in a test case.

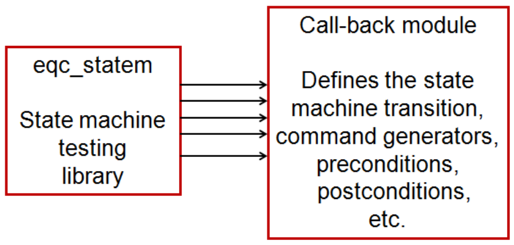

eqc_statem
State machine testing library
Call-back module
Defines the state machine transition, command generators, preconditions, postconditions, etc.

Figure 5.1: QuickCheck `eqc_statem`

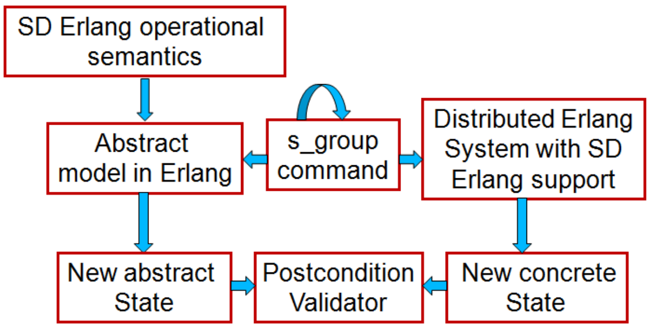


Figure 5.2: Testing SD Erlang Using QuickCheck `eqc_statem`

5.2 Validation of SD Erlang

To validate the SD Erlang semantics and implementation using `eqc_statem`, an abstract model of the SUT is needed as discussed in Section 5.1. Our approach, as illustrated in Figure 5.2, is to derive the abstract model from the semantics specification, i.e. we translate the semantics specification into an abstract model written in Erlang. The abstract model contains the abstract state representation and the transition from one state to another when an operation is applied. There is a transition function for each SD Erlang operation defined in the semantics.

An Erlang representation of the abstract state (Figure 5.3) shows a clear mapping from the formal state specification to the Erlang representation. The abstract model state is initialised according to the actual distributed SD Erlang system that we test. Thus, both the abstract state machine and the tested system have the same starting point. For our actual testing setup we use 14 SD Erlang nodes, 12 of which are normal nodes and 2 are hidden nodes.

A collection of data generators have been defined to guide the automatic generation of data value fed to s_group operations; the callback function `command/1` allows QuickCheck to automatically generate candidate test commands (Figure 5.4). The arguments supplied to each symbolic call of an s_group operation are also data generators. The `frequency/1` function makes a weighted choice between the generators in its argument, such that the probability of choosing each generator is proportional to the weight paired with it. A test case is a sequence of s_group operations, and by default, QuickCheck generates 100 test cases for each run unless the testing fails.

The conditions defined in `precondition/2` function serve as a filter, so a candidate command is included in a test case only if the preconditions for that operation are met. For instance, the precondition for `new_s_group` operation is defined as follows:

```
-record(state,
        {groups              =[] :: [group()],
         free_groups         =[] ::[free_group()],
         free_hidden_groups =[] ::[free_hidden_group()],
         nodes               =[] ::[a_node()]}).

-type group()::{s_group_name(), [node_id()], namespace()}.
-type free_group()::{[node_id()], namespace()}.
-type free_hidden_group()::{node_id(), namespace()}.
-type a_node()::{node_id(), node_type(), connections(),
                 gr_names()}.
-type gr_names()::free_normal_group|free_hidden_group|
                  [s_group_name()].
-type namespace()::[{atom(), pid()}].
-type connections()::[node_id()].
-type node_type()::visible|hidden.
-type s_group_name()::atom().
-type node_id()::node().
```

Figure 5.3: An Abstract State Representation in Erlang

```
precondition(_S, {call, ?MODULE, new_s_group,
                  [{_SGroupName, NodeIds, _CurNode},
                   _AllNodeIds]}) ->
        NodeIds/=[];   %% A new s_group cannot be empty
```

Each generated test command is applied to both the abstract model and the actual tested system. The application of *the test command to the abstract model* takes the abstract model from its current state to a new state as described by the transition functions; whereas the application of *the test command to the real system* leads the system to a new actual state. The actual s_group-related state information is collected from each tested node, then the information is merged and normalised to the same format as the abstract state representation. For a successful testing, after the execution of each test command the normalised actual state should be the same as the abstract state; this constraint, together with some other generic invariants, is specified as the postconditions of s_group operations. For instance, the postcondition for new_s_group operation is defined as follows:

```
-- Res is the actual value returned by the command
-- ActualState is the actual state information collected from nodes
postcondition(S, {call, ?MODULE, new_s_group,
                  [{SGroupName, NodeIds, CurNode}, _AllNodeIds]},
              {Res, ActualState}) ->
  {AbsRes, NewS} =
    new_s_group_next_state(S, SGroupName, NodeIds, CurNode),
  (AbsRes == Res) and is_the_same(ActualState, NewS);
```

Postconditions are validated after execution of every test command. If a validation fails the failing test case is reported.

```
command(S) ->
  frequency(
     [{5, {call, ?MODULE, new_s_group,
           [gen_new_s_group_pars(S), all_node_ids(S)]}}
     ,{5, {call, ?MODULE, add_nodes,
           [gen_add_nodes_pars(S), all_node_ids(S)]}}
     ,{5, {call, ?MODULE, remove_nodes,
           [gen_remove_nodes_pars(S), all_node_ids(S)]}}
     ,{5, {call, ?MODULE, delete_s_group,
           [gen_delete_s_group_pars(S), all_node_ids(S)]}}
     ,{10,{call, ?MODULE, register_name,
           [gen_register_name_pars(S), all_node_ids(S)]}}
     ,{10,{call, ?MODULE, whereis_name,
           [gen_whereis_name_pars(S),  all_node_ids(S)]}}
     ,{10,{call, ?MODULE, re_register_name,
           [gen_re_register_name_pars(S), all_node_ids(S)]}}
     ,{10,{call, ?MODULE, unregister_name,
           [gen_unregister_name_pars(S), all_node_ids(S)]}}
     ,{10,{call, ?MODULE, send,
           [gen_send_pars(S),all_node_ids(S)]}}
     ,{1, {call, ?MODULE, node_down_up,
           [gen_node_down_up_pars(S), all_node_ids(S)]}}
     ]).
```

Figure 5.4: Command Generator

The top-level property that connects all the parts together is defined as follows:

```
prop_s_group() ->
  ?SETUP(
    fun setup/0,
    ?FORALL(Cmds,commands(?MODULE),
            begin
              {H,S,Res} = run_commands(?MODULE,Cmds),
              teardown(),
              setup(),
              pretty_commands(?MODULE, Cmds, {H,S,Res}, Res==ok)
            end)).
```

Here, `setup/0` and `teardown/0` functions start and terminate the actual SD Erlang system respectively.

To date we have validated the nine functions listed in Figure 5.4. We found and fixed four bugs in the semantics and identified an issue in the SD Erlang implementation, i.e. in some cases after an s_group has been removed the actual s_group structure and the abstract s_group structure initially do not match each other but eventually become the same. The structures become identical after a 60s timeout and an error report from the real system. Currently, we are not sure about the origins of the issue but we believe it is due to node synchronisation, and work on resolving of the issue.

Chapter 6

Implications and Future Work

This technical report addresses SD Erlang s_group reliability and s_group semantics. The reliability was addressed by implementing a node recovery mechanism and analysing s_groups that have the same name. To identify the most suitable node recovery mechanism we explored several alternate strategies using system state estimation and analysing RELEASE SD Erlang exemplars (Chapter 2). We outline the approach we have implemented where each node keeps its own s_group information in a configuration file that is updated by the node every time the s_group configuration changes. To guarantee a high probability of s_group name uniqueness when s_groups are started dynamically we have modified function `s_group:new_s_group` to generate names that are very probably unique (Chapter 3). If duplicate s_group names do occur we have ensured correct behaviour of s_groups and nodes. We have also introduced s_group transition semantics by defining an abstract state of SD Erlang systems and presented the transitions of fifteen SD Erlang functions (Chapter 4). We have validated the consistency between the formal semantics and the SD Erlang implementation using Erlang QuickCheck testing tool (Chapter 5).

Our future plans include the following.

- *SD Erlang Semantics.* Together with the Kent team continue to work on the semantics by running more QuickCheck tests, relaxing assumptions, and the outstanding SD Erlang implementation issue.
- *SD Erlang Validation.* Validating SD Erlang by running the benchmarks discussed in Section 2.2.
- *Semi-explicit Placement.* Conduct the work outlined in [CLTG14, Chapter 5].

Bibliography

[AHJW06] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, ERLANG '06, pages 2–10, New York, NY, USA, 2006. ACM.

[BCC+12] O. Boudeville, F. Cesarini, N. Chechina, K. Lundin, N. Papaspyrou, K. Sagonas, S. Thompson, P. Trinder, and U. Wiger. RELEASE: A high-level paradigm for reliable large-scale server software. In *In Proceedings of the 13th International Symposium on Trends in Functional Programming*, volume 7829, pages 263–278. Springer, 2012.

[CLTG14] N. Chechina, H. Li, P. Trinder, and A. Ghaffari. Scalable SD Erlang computation model. Technical Report TR-2014-003, The University of Glasgow, December 2014.

[CTG+14] N. Chechina, P. Trinder, A. Ghaffari, R. Green, K. Lundin, and R. Virding. Scalable reliable SD Erlang design. Technical Report TR-2014-002, The University of Glasgow, December 2014.

[EDF10] EDF. *The Sim-Diasca Simulation Engine*, 2010. http://www.sim-diasca.com.

[IOS10] Ahmad Ali Iqbal, Maximilian Ott, and Aruna Seneviratne. Simplistic hashing for building a better bloom filter on randomized data. In *Proceedings of the 2010 13th International Conference on Network-Based Information Systems*, pages 325–331, Washington, DC, USA, 2010. IEEE Computer Society.

[Klo10] Rusty Klophaus. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, pages 14:1–14:1, New York, NY, USA, 2010. ACM.

[LMS05] P. Leach, M. Mealling, and R. Salz. A universally unique identifier (UUID) URN namespace, 2005. RFC 4122.

[PAS11] Manolis Papadakis, Eirini Arvaniti, and Kostis Sagonas. PropEr: A QuickCheck-inspired property-based testing tool for Erlang, 2011. http://proper.softlab.ntua.gr.

[Rot94] H. G. Rotithor. Taxonomy of dynamic task scheduling schemes in distributed computing systems. *IEE Proceedings Computers & Digital Techniques*, 141(1):1–10, 1994.