LocalysIt: A Strategy for Local Analysis of Networks of CSP Processes — Extended version. Technical Report

Pedro Antonino Augusto Sampaio

April, 2013

Abstract

Based on a characterisation of process networks in the CSP process algebra, we formalise a strategy of local deadlock analysis using the traces and the stable failures models of CSP. The strategy proposes simple steps for analysing acyclic networks together with a behavioural pattern approach for the verification of cyclic networks. A distinguishing feature of our approach is its mechanisation in terms of a set of refinement assertions, which can be checked by CSP tools like FDR. Moreover, a case study is introduced to demonstrate the effectiveness of our strategy, including a performance comparison using FDR to analyse the original model and another comparison with the Deadlock Checker. We also point out that our strategy can be used in a compositional way, which also applies similar strategies for local analysis, implemented as a systematic approach to system design that is deadlock free by construction.

Local Analysis, Deadlock Freedom, CSP, FDR, Behavioural pattern

Contents

Chapter 1

Motivation

For concurrent system deadlock freedom is a hard property to guarantee. For those systems two main techniques are employed for that purpose: model checking and proof based approaches using a semantic model. Although those techniques are complete in the sense that for a given deadlock free system, a proof or a verification can be achieve, they have considerable drawbacks.

The verification using model checking alone generally verifies the state space of the network as a whole. This state space exploration generally grows exponentially with the number of processes of the system. For instance, if a network has components with 10 states each, the composition of this behaviour which is the behaviour of the network can go up to 10^N, where N is the number of processes of the network. It is easy to predict that to large networks with a substantial number of processes (N), this verification can became intractable.

In the case of proving using the semantics the problem is a different one. In order to be able to perform such a proof, one has to have a deep understanding of the notation semantics and a deep knowledge of the proof system to be used. In addition, one must also have the insight leading to the proof itself in order to carry it using those models. Therefore, to use this approach one has to acquire a great body of knowledge before performing the proof.

In order to illustrate this problems consider the case of the dining philosophers. The philosophers having 7 states (thinking, picked fork one, picked fork two, eating, released fork one, released fork two, stand up) and the forks processes with 3 states (acquired by philosopher one, acquired by philosopher two, released). This system can go up to $3^{N_{Forks}} \times 7^{N_{Philosophers}}$, showing that for large systems with a large N, this system will have an intractable number of states. In the case of the proof based approach, if the notation used is CSP, one would have to learn the *stable failure* semantic model, what is a deadlock in this model and would have to find a proof demonstrating that a deadlock state is not reached, a significant complex task that cannot be fully automatized.

One alternative to this approaches is to create a hybrid technique, consisting of proving, using semantic models and a proof system, that for a particular class of well defined systems, a property can be verified by only checking a small portion of the system. This principle of guaranteeing a property by only verifying a small subset of the system is called *local analysis* and is the core principle of our strategy, which provides:

- A network model based on [1], but with notion of structure that is completely decoupled from the behaviour of network.
- A strategy of decomposition and verification also based on [1], but improved with a systematization. In addition, we defined a pattern based approach, where we defined formally a pattern similar to the ones described in [6, 4].
- An encoding of the verification part of the strategy using refinement assertions, which is a major contribution of this work. This enables the user to automatically verify the conditions imposed by the strategy to guarantee deadlock freedom in tools like FDR [15] and PAT [17].

A case study is introduced as a proof of concept of our strategy, as well as a performance comparison between our strategy, the FDR [15] deadlock freedom assertion, and a tool developed with a specific purpose of deadlock verification called Deadlock checker [16].

Chapter 2

Communicating Sequential Processes (CSP)

Communicating Sequential Processes (CSP) [9, 6, 7] is a notation used to model concurrent systems where processes interact through communications. In this notation sequential processes can be defined as a succession of events, which can in turn be combined using high level parallel operators to create complex concurrent processes.

The CSP notation has a rich high level set of operators for modelling concurrent systems. For the sake of brevity only the operators used in the work are presented; for a full account of CSP operators one should consult [7]. The basic operator used to build sequential processes is the prefix($\rightarrow$). It constructs a process from an event and another process. Process $a \rightarrow Q$ is a process that performs the event a and then behaves like process Q. In order to construct concurrent systems the parallel operators must be used. The alphabetized parallel($_{A_P}\|_{A_Q}$) operator is used to combine process where they synchronize in events in the set $A_P \cap A_Q$. For instance considering process $P \ {}_{\{a,b\}}\|_{\{b,c\}} \ Q$, it can only perform one of the events in $\{a, b\} \cap \{b, c\}$ after trace s; if P and Q are able to communicate this same event after s, in this example, event b. Also for the parallel operator the process P is only able to perform events in A_P and Q can only perform events in A_Q. The hiding operator($\setminus$) is used to abstract the behaviour of a process. Process $P \setminus \{a, b\}$ makes events a and b internal events, so the traces of process P will no longer contain events a and b. Some other operators used in the semantic models and for the tool used are introduced as they are presented in the text.

In order to reason about the notation, CSP embodies a collection of mathematical models. The classical ones are: *traces*, *stable failures*, and the *failures-divergences* model. In the first model, processes are described as a set of sequences of finite events it can perform. $traces(P)$ gives this set. In the *stable failures* model, a process is represented by its traces as previously described and by its stable failures. Stable failures are pairs (s, X) where s is a finite trace and X is a set of events that the process can refuse to do after performing the trace s. At the state where the process can refuse events in X, the process must not be able to perform an internal action, otherwise this state would be unstable and will not be taken into account in this model. The function $refusals(P, s)$

gives the set of X's that a process P can refuse after s and $failures(P)$ gives the set of stable failures of process P. Only these two models are used in this work.

Chapter 3

Network model

Systems can be seen as a set of components that interact to provide a set of functionalities. In this sense, we can model a system using the CSP notation as a network of processes, where processes represent the components of the system. This network model, similar to the one presented in [1], provides an abstract and structured body of concepts to reason about deadlocks in networks in a simpler manner.

The most fundamental concept is the one of atomic tuples, which are the building blocks of networks. Atomic tuples are structures that represents the atomic components of a system. Those are triples that contain an identifier for the component, the process describing the behaviour of this component and an alphabet that defines the events that this component can perform.

Definition 1 (Atomic tuple)**.** Let *CSP_Processes* be the set of CSP processes, Σ the set of CSP events and id an identifier of the process. A basic process P is a triple such that:

$$P \in Atomics$$

where: $Atomics \mathrel{\widehat=} \mathbb{N} \times \mathbb{P}\,\Sigma \times CSP\_Processes$

The representation of a system is then given by a set of those atomic tuples. As previously mentioned, this representation of a system is called a network and is the main object of study in this work. We use the word system and network interchangeably in the rest of this work.

Definition 2 (Network)**.** Let *Atomics* be the set of basic processes as previously described. Then a network is a set as follows:

$$V \subset Atomics$$

where V is finite.

To reason about deadlock we must give a proper definition of the behaviour of such a structure. The behaviour of the system is given as a composition of the behaviour of each component. In fact, components interact through alphabets intersection, i.e. if an atomic tuple has alphabet A_1 and another tuple has alphabet A_2, then they interact in the events $A_1 \cap A_2$. This behaviour is achieved in the CSP notation by using the alphabetised parallel operator,

where processes and alphabets are extracted from atomic tuples. To generalise our notation, we use an indexed notation for the alphabetized parallel operator. It behaves like the binary one, where processes interact in alphabet intersection. The notation $A(pid)$ and $P(pid)$, extract the alphabet and the behaviour of an the atomic process identified by pid from the network V.

Definition 3 (Behaviour of a network)**.** $B(V) = \Big\|_{pid \in \mathrm{dom}\ V} [A(pid)]P(pid)$

3.1 Deadlock analysis in networks

Our model is structured differently from the one in [1]. While in [1], they use the structure provided by the indexed alphabetised operator to deal with the set of processes, i.e. they create a intrinsic bond between behaviour and structure, in the present work we have a clear distinct structure called a network with a model which embodies a function giving the behaviour of such a structure. Even though the structural difference, the semantic of the behaviour of a network remains the same. Thus we are able to reuse some of the results obtained in [1], provided some assumptions are satisfied. The first of them is *busyness*. A busy network is a network whose atomic components are deadlock free. The second assumption is *atomic non-termination*, i.e. no atomic component can terminate. The last assumption concerns interactions. A network is *triple-disjoint* if at most two processes share an event, i.e. if for any three different atomic tuples their alphabet intersection is the empty set. A network satisfying these three assumptions is called a *live* network.

For a *live* network, a deadlock state can only arise from an improper interaction between processes, since no process can individually deadlock or terminate. This particular misinteraction is captured by the concept of *ungranted request*. An ungranted request is an attempt of one component, say pid_1, to communicate with another one, say pid_2, but pid_2 cannot offer the events expected by pid_1. In addition, the process pid_1 must not be able to perform internal actions, i.e. events that do not involve another process to synchronize. This definition is formally given in the *stable failures* model of CSP as follows.

Definition 4 (Ungranted requests)**.** Let V be a network and pid_1 and pid_2 two processes identifiers of V tuples. There is an ungranted request after a given trace s of the network from pid_1 to pid_2 if the following predicate holds.
$ungranted\_request(s, pid_1, pid_2, V) =$
$\exists\, X_1 : refusals(P(pid_1), s \upharpoonright A(pid_1)), X_2 : refusals(P(pid_2), s \upharpoonright A(pid_2)) \bullet$
$request(A(pid_1), A(pid_2), X_1) \wedge$
$ungrantedness(A(pid_1), X_1, X_2) \wedge$
$in\_vocabulary(A(pid_1), A(pid_2), X_1, X_2, Voc(V))$
where:

- $request(A_1, A_2, X) = (A_1 \setminus X) \cap A_2 \neq \emptyset$
- $ungrantedness(A, X_1, X_2) = (A \setminus X_1) \subseteq X_2$
- $in\_vocabulary(A_1, A_2, X_1, X_2, Voc) = (A_1 \setminus X_1) \cup (A_2 \setminus X_2) \subseteq Voc$

For instance, let $P(pid_1) = a \rightarrow b \rightarrow P(pid_1)$ and $P(pid_2) = c \rightarrow a \rightarrow P(pid_2)$, and s be the empty trace ($\langle\rangle$). In this case, process pid_1 has an ungranted request to pid_2, because it can only perform event a, which is in the vocabulary of the network, but process pid_2 does not offer this event after s.

Ungranted requests are the building blocks of deadlock in live networks. They are elements of a more complex structure that causes deadlocks. A cycle of ungrated requests is a necessary condition to a deadlocked state. In a deadlocked state a proper cycle of ungranted requests must arise among other conditions. A proper cycle is a sequence of different processes identifiers, C, where each element at the position i, i.e. element $C(i)$, has an ungranted request to the element at the position $i \oplus 1$, i.e. element $C(i \oplus 1)$, where $\oplus$ is addition modulo length of the cycle.

A proper cycle can in turn be divided into two categories: *conflicts* and *long cycles*. A *conflict* is a proper cycle of ungranted requests which has length 2, that is, it can arise from an improper communication of a pair of processes in a network. A network is then conflict-free if every pair of processes is free of conflicts. A *long cycle* is a proper cycle which has length greater than 2. This kind of cycle involves more than a pair of processes; for a long cycle of length 3 there is a need for verifying every combination of 3 or more processes in the network. Therefore, the task of verifying the presence of a long cycle of ungranted requests in a cycle is as difficult as the quest for a deadlocked state itself. After these definitions two fundamental theorems extracted from [1] are introduced.

Theorem 1 (In [1], Theorem 2, p. 223)**.** *Let V be a live network. If V is free of strong conflict, then any deadlocked state has a cycle of ungranted requests. If V is conflict-free then a deadlock state has a long cycle.*

A notion of decomposition enables the strategy to analyse smaller parts of the network independently and then conclude whether or not the network is deadlock free. In order to introduce a strategy for decomposition, one must first introduce the concept of *communication graph*. A *communication graph* is a representation of the topology of the network where nodes depict atomic components of the network and edges the alphabet intersection between components. A *disconnecting edge* is an edge that if removed increases the number of partition of the graph, i.e., an edge that is not part of a cycle in the communication graph. The partitions left after the removal of every disconnecting edge are called *essential components*. In [1] an important result allows one to conclude deadlock freedom from analysing only *disconnecting edges* and *essential components*.

Theorem 2 (In [1], Theorem 4, p. 226)**.** *Let V be a network with essential components $V_1, \ldots, V_k$ where the pair of processes joined by each disconnecting edge are conflict-free. Then if each V_i of the network is deadlock free, then so is V.*

With these two results it is already possible to fully verify a tree network in a local way, by checking only pairs of processes. Due to the fact that only proper cycles of length two can arise in tree networks one needs to prove that connected pairs of processes are conflict-free. This result can greatly tackle the performance for conflict-free tree networks, but cyclic networks cannot be verified locally by these methods. Moreover, if one tries to verify the freedom of cycles of ungranted requests, based on Theorem 1, this might be as complex as exploring the whole state space. Therefore, for networks with cycles a complete and local method for checking deadlock freedom is not available. The solution

for this problem presented next is based on behavioural patterns of the processes composing the network. Although this method is not complete, it covers a broad spectrum of systems and is locally verifiable.

3.1.1 Behavioural patterns to avoid deadlock

The approach for avoiding deadlock presented here is based on the rules described in [5, 6, 4]. These rule impede deadlock by preventing a cycle of ungranted requests to arise. In fact, the main goal of those rules is to provide an order on processes; if processes respect a behavioural pattern then no cycle of ungranted requests can arise.

Resource allocation pattern

This pattern can be used in a vast spectrum of systems. It can be applied to systems that in order to perform an action have to acquire some shared resources such as a lock. For instance, monitors can be modelled using this pattern and monitor-like features are available in most programming languages [8]. In this pattern the processes of a network are divided into *User* and *Resource* processes and a linear order is assumed on resource processes, called *RA order*. *Acquire* and *Release* are functions such that the event used by the user process pid_U to acquire the resource pid_R, is given by $Acquire(pid_U, pid_R)$, in the same way the release event $Release(pid_U, pid_R)$ is obtained. Two restrictions on processes must be satisfied in order to classify a process as a resource allocation process.

The first restriction is on the behaviour of a resource process. If the resource is not acquired then it must be available to all users that are able to acquire the given resource. Once acquired, it can only be released by the user process that acquired it. Hence, the behavior of a resource must be equivalent to the one of the process *Resource* described as follows.

- $Resource(pid_R, resource\_users) =$
  $\Box\ resource\_users \bullet Acquire(pid_U, pid_R) \rightarrow Release(pid_U, pid_R) \rightarrow$
  $Resource(pid_R, resource\_users)$

where *resource_users* is the set of users that can acquire the given resource, and the operator $\Box$ is the indexed external choice. This operator is indexed over a set, where the elements of this set are used to offer a set of deterministic behaviours.

The second restriction is imposed on the behaviour of the users. A user process must only be able to acquire a resource that is higher in the *RA order* $(>_{RA})$ than the last resource acquired. Therefore, the behaviour of a user process considering only events of acquisition and release of resource must be a refinement of this process.

- $User(pid_U, >_{RA}, acquired\_resources, user\_resources) =$
  $(\Box\ pidR : higher(max(acquired\_resources, >_{RA}), >_{RA}, user\_resources) \bullet$
  $Acquire(pid_U, pid_R) \rightarrow$
  $User(pid_U, >_{RA}, acquired\_resources \cup \{pid_R\}, user\_resources))$
  $\Box$
  $(\Box\ pid_R : acquired\_resources \bullet Release(pid_U, pid_R) \rightarrow$
  $User(pid_U, >_{RA}, acquired\_resources \setminus \{pid_R\}, user\_resources))$

where *acquired_resources* is the set of resources already acquired by the user process in question, *user_resources* is the set of resources that can be acquired by the user, *max* is a function that gives the maximal element of a set given an order as argument, *higher* is a function that given a set, an element and an order, filters elements of the set which are lower than the element given according to the given order.

An atomic component is defined as a resource allocation process, if its behaviour respects the respective conditions.

Definition 5 (Resource allocation tuple)**.** Let T be an atomic tuple, such that $T = (pid, P, A)$, T is a resource allocation tuple if the following predicate holds:

$$\begin{gathered} RA\_tuple(T) = pid \in Resources \Rightarrow Resource(pid, resource\_users) \equiv_{\mathrm{T}} P \vee \\ pid \in Users \Rightarrow User(pid, >_{RA}, acquired\_resources, user\_resources) \\ \sqsubseteq_{\mathrm{T}} P \upharpoonright AcquireReleaseEvents(pid) \end{gathered}$$

where: $AcquireReleaseEvents(pid)$ is the set of every event of acquisition and release of resources by process *pid*.

A network complies to the *resource allocation pattern* if: every atomic process is a *resource allocation tuple*, all processes are either a resource or a user process, there is no communication between two resource processes or two user processes and user and resource processes can only communicate in events for acquisition and release.

Theorem 3 (Resource allocation network is deadlock free)**.** *Let V be a live and conflict-free network. If all the following conditions holds:*

- $\forall\, T : V \bullet RA\_tuple(T)$
- $Resources \cap Users = \emptyset \wedge Resources \cup Users = \mathrm{dom}\ V$
- $Acquire \cap Release = \emptyset$
- $\forall\, pid_1, pid_2 : Resources \bullet A(pid_1) \cap A(pid_2) = \emptyset$
- $\forall\, pid_1, pid_2 : Users \bullet A(pid_1) \cap A(pid_2) = \emptyset$
- $\forall\, pid_1 : Users, pid_2 : Resources \bullet A(pid_1) \cap A(pid_2) \subseteq Acquire \cup Release$

then V is deadlock free.

Proof sketch. First of all, an ungranted request can only happen from a user to a resource and vice versa, since there is no interaction between two users or two resources. Secondly, an ungranted request from a user to a resource can only happen if the resource is acquired by some other user. Thirdly, an ungranted request from a resource to a user can only happen if the user has already acquired that resource. These conditions are guaranteed by the conditions imposed by the pattern.

Then, assuming that there is a cycle of ungranted requests, there must be a maximal resource in the cycle, say $C(i_{max})$. Thus the $C(i_{max} \oplus 1)$ must be a user process that has acquired this resource. Moreover, $C(i_{max} \oplus 2)$ is also a resource process lower in the $>_{RA}$ than $C(i_{max})$. Since $C(i_{max} \oplus 1)$ is making an ungranted request to $C(i_{max} \oplus 2)$, by the definition of the cycle, it is trying to acquire this resource. Thus, the user process $C(i_{max} \oplus 1)$ has the maximal resource $C(i_{max})$ and is trying to acquire $C(i_{max} \oplus 2)$, which is a contradiction concerning the pattern conditions. □

3.1.2 Routing rule

This pattern can be applied to networks that route messages. It has a wide applicability in network protocols. This pattern assumes an order on tuples identifiers of the network and imposes a single restriction on processes based on this given order. If a node communicate with a predecessor, it can only communicate with predecessors. That is, let us call $RtOrder(>_{Rt})$ the pattern given order, if pid_1 has a request to pid_2 with $pid_1 > pid_2$, then pid_1 must have requests only to pid_n with $pid_1 > pid_n$. An atomic process is a *routing component* if it respects this condition. In the trace model it is expressed as follows.

Definition 6 (Routing tuple)**.** Let pid be an identifier of a tuple in the network V. Then pid_1 is a routing tuple if the following predicate holds.

$Rt\_tuple(pid, V) = \forall\, s : traces(P(pid)) \bullet requests(s, pid, V) \frown predecessors(pid, V) \neq \emptyset \Rightarrow requests(s, pid, V) = predecessors(pid, V)$
where: $requests(s, pid, V) = \{pid_2 \mid pid_2 \in \text{dom } V \wedge pid_2 \neq pid \wedge initials(P(pid), s) \cap A(pid_2) \neq \emptyset\}$ and $predecessors(pid, V) = \{pid_2 \mid pid_2 \in \text{dom } V \wedge pid_2 \neq pid \wedge pid_2 < pid\}$

$initials(P, s)$ is a CSP semantic function that given a process P and a trace s, it gives the events of P that can occur immediately after the trace s.

A network is compliant with the routing pattern if every atomic tuple is a routing tuple. In addition, a network that is conform to this pattern is also deadlock free.

Theorem 4 (Routing network is deadlock free)**.** *Let V be a live and conflict-free network.*
If $\forall\, pid : \text{dom } V \bullet Rt\_tuple(pid, V)$ then V is deadlock free.

Proof sketch. Let V be a live conflict-free routing network. In a cycle of ungranted requests, there must be a greatest pid, according to *Rtorder* among the tuples participating, say pid_Max. This means that if pid_Max, say $C(i_{Max})$ in the cycle, must have an ungranted request to $C(i_{Max} \ominus 1)$, because $C(i_{Max}) > C(i_{Max} \ominus 1)$ and also $C(i_{Max} \ominus 1)$ must have a request to $C(i_{Max})$ by the definition of cycle of ungranted request. This contradicts the assumption of conflict-freedom, proving our result.

3.1.3 Client-server

This pattern can be applied for networks involving tuples that behave as a client-like process, i.e. engaging the communication with a server, or as aserver-like process, i.e. available for engaging with clients. The pattern assumes also an order on tuples identifiers, the $CSorder(<_{CS})$. This pattern impose a restrict more elaborated that the one presented in the routing rule. An atomic process compliant to this pattern must be present two disjoint behaviour, it can be acting as a engaged server, where it can only communicate with a single client, or as an not engaged server, i.e. acting as either an unengaged server, or as a client engadged or not. These two behaviours are recognized by a set of events, i.e. this pattern considers two disjoint sets of events, where *SE_set* is the set of events that can only be performed when the atomic process is in a server

engaged state (*SE_state*), and the *NSE_set* that is a set of events that can be performed in a not engaged server state (*NSE_state*).

Definition 7 (Disjoint behaviour restriction)**.** Let *pid* be a tuple identifier in a network V. The behaviours given by *SE_set* and *NSE_set* are disjoints if the following predicate holds.
$CS\_Disjoint(pid, NSE\_set, SE\_set) = NSE\_set \cap SE\_set = \emptyset \land (\forall\, s : traces(P(pid)) \bullet initials(P(pid), s) \subset SE\_set \lor initials(P(pid), s) \subset NSE\_set)$

Besides this, different restrictions must be applied to the atomic process when they are in either one of the states. If an atomic tuple is in a *SE_state*, then is must be able to communicate with a single predecessor, this means that a process pid_1 can only be a server to another process, pid_2, in the network if $pid_1 >_{CS} pid_2$. This restriction is given as follows.

Definition 8 (SE restriction)**.** Let V be a network and *pid* an identifier of an atomic tuple belonging to this network.
$CS\_SE(pid, V) = \forall\, s : traces(P(pid)) \bullet initials(P(pid), s) \subseteq SE\_set(pid) \Rightarrow \#requests(s, pid, V) = 1 \land requests(s, pid, V) \subseteq predecessors(pid, V)$
where: $requests(s, pid, V) = \{pid_2 \mid pid_2 \in \text{dom } V \land pid_2 \neq pid \land initials(P(pid), s) \cap A(pid_2) \neq \emptyset\}$ and $predecessors(pid, V) = \{pid_2 \mid pid_2 \in \text{dom } V \land pid_2 \neq pid \land pid >_{CS} pid_2\}$

The other condition applies on processes in a *NSE_states*. This condition is the same as the one presented for the routing tuples; if a process in a *NSE_state* can communicate with a predecessor then it must be able to communicate with every predecessor according to the *CSorder*.

Definition 9 (NSE restriction)**.** Let V be a network and *pid* an identifier of an atomic tuple belonging to this network.
$CS\_NSE(pid, V) = \forall\, s : traces(P(pid)) \bullet initials(P(pid), s) \subseteq NSE\_set(pid) \land requests(s, pid, V) \frown predecessors(pid, V) \neq \emptyset \Rightarrow requests(s, pid, V) = predecessors(pid, V)$
where: $requests(s, pid, V) = \{pid_2 \mid pid_2 \in \text{dom } V \land pid_2 \neq pid \land initials(P(pid), s) \cap A(pid_2) \neq \emptyset\}$ and $predecessors(pid, V) = \{pid_2 \mid pid_2 \in \text{dom } V \land pid_2 \neq pid \land pid >_{CS} pid_2\}$

The last restriction imposed by this pattern is on *SE_states*. If a process in a *SE_state* has an ungranted request to another process, this other process must be also in a *SE_state*

Definition 10 (SE ungranted restriction)**.** Let V be a network and *pid* an identifier of an atomic tuple belonging to this network.
$CS\_SE\_ungranted(pid, V) = \forall\, s : traces(P(pid)) \bullet initials(P(pid), s) \subseteq SE\_set(pid) \land \exists\, pid_2 : \text{dom } V \bullet ungranted\_request(s, pid_1, pid_2, V) \land pid \neq pid_2 \Rightarrow initials(P(pid_2), s) \subseteq SE\_set(pid_2)$

Therefore, a tuple that conforms to this pattern must respect all this conditions.

Definition 11 (CS tuple)**.** Let V be a network and *pid* an identifier of an atomic tuple belonging to this network, *NSE_set* and *SE_set* two sets of events partitioning the behaviour of the process *pid*.

$CS\_tuple(pid, V) = CS\_Disjoint(pid, NSE\_set, SE\_set) \wedge CS\_SE(pid, V) \wedge$
$CS\_NSE(pid, V) \wedge CS\_SE\_ungranted(pid, V) land(NSE\_set \cup SE\_set = A(pid))$

A network conforms to this pattern must have all atomic tuples conform the *CS_tuple* condition. In this case a network, which is conflict-free and live, is deadlock free.

Theorem 5 (CS network is deadlock free)**.** *Let V be a live conflict free network. If $\forall\, pid : \text{dom}\ V \bullet CS\_tuple(pid, V)$ then V is deadlock free.*

Proof sketch. Let V be a live conflict-free routing network. A cycle of ungranted request is composed either of processes in a *NSE_state* or in a *SE_state*, because of the *SEungrantedrestriction*. In the case of a *NSE_state*, we fall into the same case as the routing pattern, since in this state it must respect the same conditions as the routing pattern, due to the *NSE_restriction*. In a cycle composed of processes in a $NSE - state$, the cycle must have a maximal element, say $C(i_{max})$, to which process $C(i_{max} \ominus 1)$ is making a request, due to the definition of the cycle of ungranted requests. By the *SErestriction*, $C(i_{max} \ominus 1)$ must have a request to a predecessor, but we established that $C(i_{max})$ is the maximal element of the cycle, a contradiction that proves our theorem.

Chapter 4

ComposIt strategy

For concurrent system deadlock freedom is a hard property to guarantee. For those systems two main techniques are employed for that purpose: model checking and proof based approaches using a semantic model. Although those techniques are complete in the sense that for a given deadlock free system, a proof or a verification can be achieve, they have considerable drawbacks.

The verification using model checking alone generally verifies the state space of the network as a whole. This state space exploration generally grows exponentially with the number of processes of the system. For instance, if a network has components with 10 states each, the composition of this behaviour which is the behaviour of the network can go up to 10^N, where N is the number of processes of the network. It is easy to predict that to large networks with a substantial number of processes (N), this verification can became intractable.

In the case of proving using the semantics the problem is a different one. In order to be able to perform such a proof, one has to have a deep understanding of the notation semantics and a deep knowledge of the proof system to be used. In addition, one must also have the insight leading to the proof itself in order to carry it using those models. Therefore, to use this approach one has to acquire a great body of knowledge before performing the proof.

In order to illustrate this problems consider the case of the dining philosophers. The philosophers having 7 states (thinking, picked fork one, picked fork two, eating, released fork one, released fork two, stand up) and the forks processes with 3 states (acquired by philosopher one, acquired by philosopher two, released). This system can go up to $3^{N_{Forks}} \times 7^{N_{Philosophers}}$, showing that for large systems with a large N, this system will have an intractable number of states. In the case of the proof based approach, if the notation used is CSP, one would have to learn the *stable failure* semantic model, what is a deadlock in this model and would have to find a proof demonstrating that a deadlock state is not reached, a significant complex task that cannot be fully automatized. Network decomposition and behavioural patterns are concepts used in our strategy that are proved to be correct using the proof based method; and the conditions to decompose a system and to verify if a system is compliant to a pattern can be encoded in terms of refinement assertion to be checked by a model checker. Our strategy avoids the state explosion problem by only performing local verifications, i.e. it verifies only small parts of the system, instead of the entire system, to assure deadlock freedom. The problems following from the use of the proof based ap-

proach are also tackled, as the user of our strategy must only be familiar with the notions and conditions proposed in the strategy.

The strategy is composed of three steps. If one of the steps is not valid, the network must be redesigned until the steps are all validated. The steps are the following.

- Verify if the network is live.
- Verify if the network is conflict-free.
- Verify adherence of essential components (those not formed of a single atomic tuple) to a behavioural pattern.

As the strategy deals only with live networks, the first step must verify if the network is live. The verifications to assure that a network is live are all local. Deadlock freedom and non-termination must be checked for individual atomic processes, and triple disjointness for alphabets of each triple of atomic components. If a network is not live, then either some redesign must be performed in order to make the network live or this strategy is not applicable.

In the second step, the network must be verified for conflict freedom. This verification is performed by verifying if each pair of communicating processes is conflict free. This verification is also local, since only pairs of processes in parallel must be checked. If a pair of processes is not conflict free then a redesign of the network must be worked out.

The last step is applied only for essential components that are cyclic, i.e. the ones that have more than a single atomic process. In order to perform this step one needs to build the communication graph and identify the essential components. After identifying the essential components, a pattern compliance verification must be carried out. This step is vacuously valid if a network is a tree. In this case, all atomic tuples are essential components, therefore, there are only single atomic components. If a network is not a tree then there must be at least one essential component that is cyclic, i.e. composed of at least two atomic components. In this case the only applicable local technique to assure deadlock freedom is to make this component compliant to a behaviour pattern and, in the context of this paper, to the resource allocation pattern. If it is not, the essential component should be redesigned to became compliant.

The correctness of our approach is a direct consequence of the three presented steps. If all verifications are valid, the network is live, conflict free and the cyclic essential components are compliant to a pattern. This means that all essential components are deadlock free and all disconnecting edges are conflict-free. In this case, by Theorem 2, we can guarantee that our strategy assures deadlock freedom.

An important consideration to be made is that our strategy considers networks as white box structures, where one can perform more fine-grained verifications in single atomic components or pairs of components instead of the network as a whole. On the other hand, atomic processes are treated as black boxes, no fine-grained verification can be done on smaller parts of processes. Therefore, one must be careful while using the strategy not to define very complex atomic components, because, in this case, even though local analysis is performed the state explosion problem can still occur for atomic components.

4.1 Mechanisation of the strategy

One of the contributions of this work is the mechanisation of the verification of the side conditions associated with the three steps of the strategy. Conditions such as conflict-freedom and the compliance of a process to the resource allocation pattern are encoded as refinement assertions in CSP_M [15]. This has also been done for the other patterns, as can be found in [**?**]. Therefore, one can use the strategy together with tool support provided by a refinement checker, such as FDR [15].

The first important condition in our strategy is the verification of conflict-freedom for a pair of processes. A pair of process, which is atomic deadlock free and non terminating is conflict free if the alphabetised parallel composition is deadlock free. This deadlock verification captures conflict-freedom because if a conflict arises then a deadlock state is reached. The deadlock occurs because of the mutual dependence caused by the cycle of ungranted requests. This property can be translated to the following FDR refinement assertion. Let $(pid_1, P(pid_1), A(pid_1))$ and $(pid_2, P(pid_2), A(pid_2))$ be a pair of atomic tuples.

```
assert P(pid1) [A(pid1)||A(pid2)] P(pid2) :[deadlock free [F]]
```

4.1.1 Resource allocation adherence

Before presenting the second group of assertions that verifies the compliance of a process to the resource allocation pattern, we present the two auxiliary processes, part of the assertions, that represent the expected behaviour for user and resource processes. The first process is the description of the behaviour expected by a resource process in CSP_M; this behaviour was presented in the pattern description as the *Resource* process. The second process describes the behaviour expected by a user process in CSP_M. In the pattern description this specification was given by process *UserCondition*.

```
Resource(pidR,resource_users) = [] pidU :resource_users @
    Acquire(pidU,pidR) -> Release(pidU,pidR) ->
        Resource(pidR,resource_users)

UserCondition(pidU,resources_acquired, order) =
let
 user_resources = ResourcesU(pidU)
within
    ([] pidR :
       higher(max(resources_acquired,order),order,user_resources)@
        Acquire(pidU,pidR) ->
            UserCondition(pidU,resources_acquired^<pidR>,order))
     []
    ([] pidR : set(resources_acquired) @
        Release(pidU,pidR) ->
      UserCondition(pidU,removeR(pidR,resources_acquired), order))
```

The following pair of refinement assertions are used to verify if a resource is trace equivalent to the *Resource* process and the third assertion is used to verify

whether a user process refines the *UserCondition* process. Those conditions are the behavioural restrictions imposed by the resource allocation pattern on atomic tuples.

```
assert P(pidR) [T= Resource(pidR,UsersThatCanAquire)
assert Resource(pidR,UsersThatCanAquire) [T= P(pidR)

assert UserCondition(pidU,resources_acquired, order) [T= P(pid)
```

4.1.2 Routing adherence

The second group of assertion is to verify the compliance of a tuple with the routing pattern. First of all, lets introduce the technique used for pattern compliance. We introduce a controller process between two versions of the process we shall check, the *middle* process. Let us call one version *left* and the other *right*. The left version behaves without restrictions and is used only as manner to obtain the current state of the process. The middle process, the introduced process, will synchronize on any action of the left process. It will then check the state of the process on the left and impose the pattern restriction on the right version. If a restriction is not satisfied, the middle process deadlocks and the whole combination deadlocks as well, as both left and right can only perform event in synchronization with the middle process.

This technique is used because we must, for a given state of a process, both enquire about the process' state and impose a restriction on the same state, but, because as we enquire a process synchronizing in an event, after an enquiry the process state changes, thus the restriction must be imposed in a state that is previous to the current one for the process. Hence, the solution is to have two versions in the same state, left and right versions, so as to be able to enquiry the state of the left version and apply the restriction in the right version. In addition, if the restriction passes on the right side, we must guarantee that both left and right processes are in the same state. If right and left side are not in the same state, then they are useless. We achieve this by using the middle process to enquire the left version's state, impose restriction on the right version and to maintain the state consistency between the left and right version.

For the Routing pattern, the restriction imposed on patterns is that if it communicates with a predecessor, it must be able to communicate with all predecessors. The process introduced next performs exactly this check. It will synchronize on the alphabet of the process *pid*, given by the local variable, `alpha'`, and if the event performed by the left version, `am` in the process body, is a predecessor communication, then the middle process will offer a non-deterministic choice of predecessor communication to the right process. If the right process cannot perform communications with every predecessor, than there will be a deadlocked state, for the combination, left-midle-right process. If an event of communication with a successor is performed than no restriction is imposed on the right version.

```
MiddleProcess(pid) =
let
    alpha' = A(pid)
    AInf = inter(alpha',Union({A(pred)| pred<-predecessors(pid)}))
```

```
within
       [] am : alpha' @ am ->
           (if member(am,AInf) then
                |~| bm : AInf @ modify(bm) ->
                    (if bm == am then
                        MiddleProcess(type,pid)
                    else
                        DF)
           else
               (modify(am) -> MiddleProcess(type,pid)
               []
               DF))
```

where: DF is the deadlock free process. and modify is a function that takes an event and returns a fresh event. This function is needed to make the synchronization of the middle and right processes independent, by taking new fresh events, of the synchronization of the middle and left processes.

The following process is the combination left-middle-right used in te assertion for the pattern adherence. As explained, it is an parallel composition of the left process with the middle process synchronizing in the alphabet of the left process, the resulting process is put in parallel with the right version with a rename construct to make the alphabet of the left and right versions disjoints, and synchronizing in the renamed alphabet.

```
Routing_tuple(pid) =
let
P = P(pid)
alpha' = A(pid)
alphaModified = {modify(a) | a <- A(pid)}
within
(P [|alpha'|] MiddleProcess(pid))
[|alphaModified|]
P [[ a <- modify(a) | a <-alpha']]
```

The assertion for the verification of the pattern adherence to the routing pattern uses the FDR built-in deadlock assertion. As mentioned, if the process is not deadlock free, then there is a state that does not respect the patterns restriction.

```
assert Routing_tuple(pid) :[deadlock free [F]]
```

4.2 Client-Server adherence

The client-server pattern adherence is verified using the same strategy as the one presented in the routing pattern, i.e. there is a middle process, which regulates communication, receiving messages from a left version, of the process being tested, for trace control and imposing the pattern's restrictions to the right process.

The verification of pattern conformance is split into four assertions, the first one concerns the *SErestriction* already described. In this assertion, as

mentioned, we use the middle process strategy. The left process is used to keep track of the possible communications made by the process in order to say whether it is in a *SE_state*, if it is in a *SE_state* the middle process will impose the *SE_restriction* on the right version of the process.

```
MiddleSE(pid,SE_set,predecessors,successors) =
    let
        AInf = Union({ inter(A(n),A(pid)) | n <- predecessors})
        ASup = Union({ inter(A(n),A(pid)) | n <- successors})
    within
        [] a : A(pid) @ a ->
            (if member(a,inter(AInf,SE_set)) then
                ([] b : AInf @ modify(b) ->
                    (if b == a then
                        MiddleSE(pid,SE_set,predecessors,successors)
                    else
                        DF))
                []
                ([] c : ASup @ modify(c) -> STOP)
            else
                modify(a) ->
                   MiddleSE(pid,SE_set,predecessors,successors))
```

The `MiddleSE` process enquire the right process by synchronizing on the alphabet of the process, `A(pid)`. If the event is from the *SE_set* and from a predecessor, then the middle process offer all events to the right version, but with different after synchronization behaviours. If the right process synchronize in a event from a predecessor that is the same as the one performed by the left version, then the middle process guaranteed that both left and right versions are in the same state, hence, it continues proceeding to the next state. If the right version communicates with a predecessor but do not perform the same event, a valid behaviour according to the restriction then the middle process will behave like `DF` a deadlock free process that do not communicate with other processes. If the right version is able to communicate with a successor, violating the restriction, the it will synchronize with the successor events offered by the middle process and then middle process will deadlock causing the entire combination, left-middle-right to deadlock.

The combination left-middle-right is given by the following process, where it takes a pid and extract the behaviour of the process, and compose two version of the process' behaviour and the middle process. As mentioned before, we have a bijection from the alphabet of the process being tested for a new fresh alphabet, so as to have disjoint alphabets making middle communicate with the left and right version in distinct events.

```
CheckSE(pid,predecessors,successors) =
let
alphaModified = {modify(a) | a <- A(pid)}
within
(P(pid) [|A(pid)|]
MiddleSE(A(pid),SE_set(pid),predecessors,successors)) [|alphaModified|]
(P(pid) [[ a <- modify(a)  | a <- A(pid)]])
```

where: `modify(e)` is a bijection between events of a process' alphabet and a fresh alphabet.

The assertion to verify this restriction is given next, as mentioned, it checks if the combination left-middle-right deadlocks, because it will only deadlocks if a state that violates the restriction is reached.

```
assert CheckSE(pid,predecessors,successors) :[deadlock free [F]]
```

The next restriction addressed is the *NSErestriction*, we use the same middle process as well. In this restriction we have the left process used to verify if the current state of the process being tested is a *NSE_state*, if that is the case a restriction is applied by the middle process on the left process.

```
MiddleNSE(pid,NSE_set,predecessors,successors) =
let
alphaInf = Union({ inter(A(n),A(pid)) | n <- predecessors})
alphaSup = Union({ inter(A(n),A(pid)) | n <- successors})
within
[] a : A(pid) @ a -> (if member(a,inter(alphaInf,NSE_set)) then
(|~| b : predecessors @
( [] e : inter(A(pid),A(b)) @ modify(e) ->
(if e == a then
MiddleNSE(A(pid),NSE_set,predecessors,successors)
else
DF)))

[]
([] c : alphaSup @ modify(c) -> STOP)
else
modify(a) -> MiddleNSE(A(pid),NSE_set,predecessors,successors)
[]
DF
)
```

In the same way, as for the previously introduced middle process, `MiddleNSE` synchronizes with the left version of the process being tested, if the synchronized event is a *NSEevent*, then the process is in a *NSEstate*, and the middle process will impose a condition on the right version of the process. The middle process will offer, for each predecessor of the process being tested, a non-deterministic choice of the whole set of events, that is, if the right version can perform at least one event of communication with every predecessor then no deadlock arise, otherwise,the case that violates the condition, a deadlock state is reached. If the right process also can communicate with a superior, then it also violates the restriction and a deadlock state is reached. Not that if the initial communication is not a *NSEevent*, then no restriction is imposed.

The combination left-middle-right is given by the following process. In this case, we also have a bijection from the alphabet of the process being tested to a fresh alphabet, in order to enable a disjoint synchronization between the left and right version of process in question.

```
CheckNSE(pid,predecessors,successors) =
```

```
let
alphaModified = {modify(a) | a <- A(pid)}
within
(P(pid) [|A(pid)|] MiddleNSE(A(pid),NSE_set(pid),predecessors,successors))
[|alphaModified|] P(pid) [[ a <- modify(a)  | a <- A(pid)]]
```

The assertion for *NSErestriction* conformance is given also as a deadlock verification, since it will only deadlock if a state that not respect the restriction is reached.

```
assert CheckNSE(pid,predecessors,successors) :[deadlock free[F]]
```

The next assertion implements the *SEungrantedrestriction*, which is encoded also using the middle process strategy. In this restriction, we use the left and right version with the same purpose as the other previous assertions.

```
MiddleSEUng(pid1,pid2,SE_set_pid1,NSE_set_pid2) =
let
alphaComm = inter(A(pid1),A(pid2))
alphaCommModif = { modify(e) | e <- alphaComm }
within
[] a: alphaComm @ a ->
([] b : alphaComm @ modify(b) -> (
if a == b then
MiddleSEUng(A(pid1),A(pid2),SE_set_pid1,NSE_set_pid2)
else
if member(a,SE_set_pid1) and member(b,NSE_set_pid2) then
STOP
else
DF
))
```

The `MiddleSEUng` process verifies if the left and right processes are in a *SE_state*, if that is the case then it verifies if it can also perform an event that is a NSE_event for the peer process, which pid is passed as parameter (`pid2`), if it can perform such an event, then the process might have an ungranted request to the its peer. Note that this restriction implemented is slightly more restrictive than the *SEungrantedrestriction* discussed in the previous chapter. It was implement this way, because cheaper in terms of performance. The assertion that captures the exact restriction should take two behaviour as parameter and check N^2 number of states, in our case, only N number of states are checked. This restriction implemented implies the *SEungrantedrestriction*, because a process that satisfies this restriction cannot have an ungranted request from a SE_state to another process in a NSE_state, since it is not able of performing its own SE_events and NSE_events of a peer process in the same state.

The combination left-middle-right is given by the following process.

```
CheckSEUng(pid1,pid2) =
let
alphaComm = inter(A(pid1),A(pid2))
alphaCommModif = { modify(e) | e <- alphaComm }
```

```
within
(P(pid1) [|alphaComm|]
MiddleUngBToB(A(pid1),A(pid2),SE_set(pid1),NSE_set(pid2))
[|alphaCommModif|] P(pid1) [[ a <- modify(a)  | a <- alphaComm]]
```

As usual in our strategy a deadlock state is reached if the restriction is not satisfied. Therefore, the assertion for *NSEungrantedrestriction* verification is given as follows.

```
assert CheckSEUng(pid1,pid2) :[deadlock free [F]]
```

The disjoint restriction is not implemented yet, but is just a matter of effort that has not been put yet. We can easily implement using the middle process strategy.

At this point, one might have noticed that none of the conditions involving set operations are presented. Although FDR is a very efficient model checker, it is not designed for performing exhaustive set operations. The conditions involving set operations can also be mechanised, for instance, using a constraint solver. This ia simple task and out of the scope of this paper.

Chapter 5

Case study

As a proof of concept of our strategy, a case of study was conducted, the dining philosophers. The LocalysIt strategy was used to analyse a network of philosophers and forks. In the case study, we present a simple version with three forks and three philosophers, but the analysis was conducted with up to 20000 philosophers and forks. First of all, the processes representing philosophers and forks are introduced in the CSP notation. Concerning the philosopher behaviours, *Phil_live* is an asymmetric version of the philosopher and *Phil_dead* a symmetric one. For the sake of brevity only single examples of assertions are introduced, for a full account the reader should refer to D.

```
Phil_live(y) =
    if y != MAX-1 then
        think.y -> pickup.y.y -> pickup.y!next(y) ->
        putdown.y.y -> putdown.y!next(y) -> eat.y -> Phil_live(y)
    else
        think.y -> pickup.y!next(y) -> pickup.y.y ->
        putdown.y.y ->putdown.y!next(y) -> eat.y -> Phil_live(y)

Phil_dead(y) = think.y -> pickup.y.y -> pickup.y!next(y) ->
    putdown.y.y -> putdown.y!next(y) -> eat.y -> Phil_dead(y)

Fork(forkid) =
    let
        x = (forkid-MAX)
    within
        pickup!prev(x)!x ->  putdown!prev(x)!x -> Fork(forkid)
        [] pickup!x!x -> putdown!x!x -> Fork(forkid)
```

The network of forks and philosophers is given as follows. The first two sets are the networks of asymmetric and symmetric philosophers, respectively. The third set is the network of forks and the following two networks are the dining philosophers asymmetric and symmetric.

```
Phils_set_live = {(pid,Phil_live(pid), alpha_phil(pid))|pid<-Users}
```

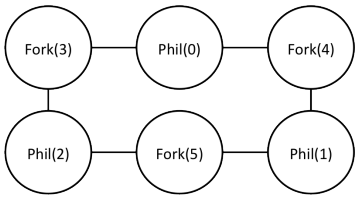

Figure 5.1: Communication graph of dinning philosophers with 3 philosophers and 3 forks.

```
Phils_set_dead = {(pid,Phil_dead(pid), alpha_phil(pid))|pid<-Users}

Forks_set = { (pid,Fork(pid), alpha_fork(pid)) | pid<-Resources}

Network_alive = union(Forks_set,Phils_set_live)
Network_dead = union(Forks_set,Phils_set_dead)
```

After defining our working model, the structure of networks that we want to check, we are ready to apply the strategy. The first step is the verification that our network is live. In order to be live a network must be busy, atomic non-terminating and triple-disjoint. For busyness and atomic non-termination, FDR implements a deadlock freedom verification that checks both conditions. Since only behavioural conditions are mechanised, triple disjointness is not automatically verified. Hence, for this condition we just assume that our design is triple disjoint as it indeed is. Therefore, for each fork and philosopher we check if it is deadlock free using the following assertions.

```
assert Fork(3) :[deadlock free [F]]
assert Phil_live(0) :[deadlock free [F]]
assert Phil_dead(0) :[deadlock free [F]]
```

After this step our network is indeed live and we go to the next step, which consists of verifying that every pair of communicating processes of our network is conflict-free. For this purpose, we use the assertion defined in the previous section. Therefore, for each pair of philosopher and fork that communicate, we check the following.

```
Pair(pidU,pidR) =
     Fork(pidR) [alpha_fork(pidR)||alpha_phil(pidU)] Phil_live(pidU)
assert Pair(0,3) :[deadlock free [F]]
```

This assertion also holds, since every pair of processes is conflict free for both the asymmetric and the symmetric cases, then we are able to pass to the last step. In the last step, we need to build the communication graph in order to find the essential components of our network. Since the dining network is a ring, it is a single essential component composed of six atomic components; the communication graph is depicted in Figure 5.1. This means that the resource allocation pattern must be applied for the network as a whole, due to its cyclic topology. Therefore, we need to verify that both philosopher and fork processes are complying with the pattern. We, as users of the strategy, are entitled to give

the order and the sets of resources and users in order to be able to verify if our network conforms to the resource allocation pattern. As a matter of simplicity and efficiency, we chose the order to be the order given by the operator greater than ($>$) on resource identifiers. Because of this, we do not pass an order as an argument and use the CSP built in operator in the *UserProcess*. The set of resources is given by the forks processes and the set of users is given by the set of philosophers. As forks are resource processes, fork processes must be trace equivalent to the *Resource* process. For philosophers, they must obey the restriction imposed by the process *UserCondition*, that is, their projection on events of acquisition and release must be a refinement of the *UserCondition* process. These conditions are given by the following assertions.

```
assert Fork(3) [T= Resource(3)
assert Resource(3) [T= Fork(3)

assert UserCondition(0,<>, <>) [T= Projection_phil_live(0)
Projection_phil_live(0) =
    Phil_live(0) \ diff(Events,union(AcquireU(0),ReleaseU(0)))
```

In this last step, the last assertion above holds only for the asymmetric dinning which is represented by the *Network_alive*. The Philosopher with $pid = 2$ in the symmetric case is not compliant with the pattern. This happens because, according to its behaviour, it picks first the Fork number 5 and then the fork number 3, which violates the condition imposed by the *UserCondition* process. Therefore, this symmetric network must be redesigned conform to the pattern.

5.1 Performance analysis

As one of the main advantages of using the strategy is to locally verify networks, some performance gain is expected. In order to assess the performance gains, a set of scripts were created to generate instances with a parametrized number of processes of the dinning philosopher model and the set of assertions implementing the strategy. The experiment scripts can be found in [14] or the appendices.

For the experiment, we used a Python [19] based template language, called Cheetah [18]. A parametrized instance of the dinning philosopher was created using a Cheetah template, that can be found in Appendix A. In addition, a set of Python scripts were created to: generate an instance of the problem with a given number of philosopher using the Cheetah template; execute FDR in a batch mode, passing the generated instances as parameters; and generate a set of log files, they can be found in Appendix B. The *Config.py* is a property file to configure environment variables such as FDR home, and the folder in which the files of the experiment are. The *TemplateCSP.py* is a file that is responsible for charging the *phil_div.template*, so as to be used by the *Generate.py*. The *Generate.py* script is responsible for generating the files to be run by FDR. It will create a *Gerados* folder where the generated files are copied. The *Execute.py* is resposible for taking the generated files and running in FDR. As mentioned all these scripts can be found in the Appendix B. Also an example of a generated file is provided in Appendix C.

# of processes	Deadlock checker	AnalysIt	FDR
10	0.03	0.25	0.087
20	0.05	0.54	412
100	0.26	2.46	*
200	1	5.07	*
1000	7	25.8	*
2000	22	59.4	*
10000	56	330	*
20000	**	672	*

$*$ Exceed the execution limit of 1 hour

$**$ Tool does not support the # of processes

Table 5.1: Performance comparison measured in seconds.

The experiment was conducted using a 2.2 GHz Intel Core i7, with 8 GB 1333 MHz DDR3 of RAM and a SATA disk in an OS X 13.8.3 operating system. Where we run the experiment 2 times and took the average of both times.

In our experiment, three approaches were compared. The first one, was to use a tool that was specifically designed to verify deadlock freedom, the Deadlock Checker [16]. The second approach was to use our strategy with the conditions being verified by FDR [15]. The last approach was to use FDR's built-in assertion for deadlock verification of the entire model. In Table 1, there is a synthesis of the results obtained.

The results demonstrate how the time for deadlock verification can grow exponentially with the linear increase of number of processes. FDR's assertion is a global method for the verification of deadlock freedom and due to the exponential explosion of the number of states to check it exceed the limit of 1 hour of execution, established for the experiment, with 100 processes in parallel. The Deadlock checker is the most efficient method but it only allows to check a network with up to 10000 processes. Finally, our strategy has a comparable efficiency, and it grows linearly with the increase of the number of processes. A consideration about performance is that the verifications of our strategy were segmented into a set of files that could be verified in parallel, since we have realised that FDR presents a better performance when a large number of assertions to be checked are split in several files. This means that the result presented in Table 1, for our strategy, could be reduced to the time presented divided by the number of files to be checked if a number of processor cores equivalent to the number of files were available.

Chapter 6

Related work

Roscoe and Brookes developed a structured model for analysing deadlock in networks [1]. They created the model based on networks of processes and a body of concepts that helped to analyse networks in a more elegant and abstract way. Roscoe and Dathi also contributed by developing a proof method for deadlock freedom [2]. They have built a method to prove deadlock freedom based on variants, similar to the ones used to prove loop termination. In their work, they also start to analyse some of the patterns that arise in deadlock free systems. We based the approach on these ideas, and developed a systematic and mechanised strategy for local analysis. Although their results enable one to verify locally a class of networks, there is no framework available to use their results such as the one presented here. Also, their definition of the network structure and behaviour were intrinsically linked. In this work we provide a structure decoupled from its behaviour. This separation of concerns allow us to enhance this structure in the future without losing the results already obtained for the behavioural analysis of networks.

A more recent work of Roscoe et all [11] presenting some compression techniques used for FDR are used to check the same example for up to 10^{100} processes. Compression techniques are an important complementary step for further improving our strategy.

Following these initial works, Martin formally defined some design rules to avoid deadlock freedom [4]. He also developed an algorithm and a tool with the specific purpose of deadlock verification, the Deadlock checker [16]. This tool is very efficient, as already mentioned, but it has a different analysis perspective when contrasted with LocalysIt. The Deadlock checker reduces the problem of deadlock checking to the quest of cycles of ungrated requests, in live networks, whereas LocalysIt verifies whether a system complies with a pattern, an entire local method. This reflects in the counter example generated in the case of a system that cannot be guaranteed to be deadlock free for those techniques. In the Deadlock checker an atypical behaviour of the system is presented, i.e. a possible cycle of ungrated requests is presented and the user is responsible for analysing the global network looking for the possible causes. In localysIt, the conditions will fail to atomic processes which do not comply to a pattern. Hence, the user only needs to deal with the complexity of analysing a single process at a time.

In a recent work Ramos developed a strategy to compose systems guaran-

teeing deadlock freedom for each composition [12]. The main drawback with his method is the lack of compositional support to cyclic networks. With the rules presented one is able to, in a compositional way, connect components in order to build a tree topology component. He presented a rule to deal with cyclic components but it is not compositional, in the sense that the verification of its proviso is not local, i.e. it must be performed in the system as a whole. Our strategy complements and can be easily combined with this compositional approach. This would result in a sound and constructive approach to (local) deadlock analysis. Exploring this combination is actually a major topic for future work.

Chapter 7

Conclusion and future work

Our verification strategy focuses on a local analysis of deadlock freedom of design models of concurrent systems which obey certain architectural patterns. Although this method is not complete, it already covers a vast spectrum of systems, those are acyclic systems conflict free and cyclic systems, that can be designed in terms of one of the formalised patterns. The strategy seems promising in terms of performance, applicability and complexity mastering, nevertheless some improvements are apparent.

The first improvement is the mechanisation of the conditions involving set operations using a proper tool. A major improvement to be considered is the use of compression techniques to enhance performance of local checks. The LocalysIt strategy can benefit from the compressing techniques already implemented in FDR [11, 15].

An interesting aspect to consider are the systems with asynchronous middlewares. There has been some studies in buffer tolerance aspects that could be unified in our strategy to enable the verification of buffered systems by verifying their bufferless analogues [13]. This could be very useful in modeling and verifying services and distributed programs.

A particular interesting connection to be established as future work, as already mentioned, is the integration of the LocalysIt strategy and the one presented by Ramos in [12]. A distinguishing feature of our strategy is precisely the possibility of combining it with other systematic approaches to analysis.

Finally, we plan to develop further case studies and carry out performance analysis based on other patterns than the one explored in this paper.

Bibliography

[1] Brookes, S.D., Roscoe, A.W.: Deadlock analysis in networks of communicating processes. Distributed Computing. 4, 209–230 (1991)

[2] Roscoe, A.W., Dathi, N.: The pursuit of deadlock freedom. Information and Computation. December 3, 289–327 (1987)

[3] Mota, A., Sampaio, A.: Model-checking CSP-Z: Strategy, tool support and industrial application. Science of Computer Programming. 40, 59–96 (2000)

[4] Martin, J.M.R., Welch, P.H.: A Design Strategy for Deadlock-free concurrent systems. Transputer Communications. 3(3), 1–18 (1997)

[5] Dijkstra, E.W.: A Class of Simple Communication Patterns. Selected Writings on Computing : A Personal Perspective. Springer-Verlag (1982)

[6] Roscoe, A. W.: The theory and practice of concurrency. Prentice Hall (1998)

[7] Roscoe, A. W.: Understanding concurrent systems. Springer-Verlag (2010)

[8] Buschmann, F., Henney, K., Schmidt, D. C.: Pattern-Oriented Software Architecture, Volume 4: A Pattern Language for Distributed Computing. Wiley (2007)

[9] Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall (1985)

[10] Brookes, S.D., Roscoe, A. W.: An improved failures model for communicating processes. In: Proceedings of the Pittsburgh seminar on concurrency, 197, 281–305. Springer LNCS (1985)

[11] Roscoe, A. W., Jackson, D.M., Gardiner, P.H.B., Goldsmith, M.H., Hulance J.R., Scattergood, J.B.: Hierarchical compression for model-checking CSP or How to check 10^{20} dining philosophers for deadlock. In: Proceedings of the First International Workshop on Tools and Algorithms for Construction and Analysis of Systems, 135–152. Springer-Verlag (1995)

[12] Ramos, R., Sampaio, A., Mota, A.: Systematic development of trustworthy component systems. In: 2nd World Congress on Formal Methods, 5850, 140–156. LNCS Springer (2009).

[13] Roscoe, A. W.: The pursuit of buffer tolerance. Unpublished draft. `http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/106.pdf` (2005)

[14] LocalysIt case study files, http://www.cin.ufpe.br/~prga2/tech/experiment.zip

[15] FDR User manual, version 2.94, University of Oxford, http://www.cs.ox.ac.uk/projects/concurrency-tools/fdr-2.94-html-manual/index.html

[16] Deadlock Checker web repository, http://wotug.org/parallel/theory/formal/csp/Deadlock/

[17] PAT Toolkit web repository, http://www.comp.nus.edu.sg/~pat/

[18] Cheetah template language web repository, http://www.cheetahtemplate.org/

[19] Python language web repository, http://www.python.org/

Appendix A

Cheetah template

phil_div.template: Cheetah template used to generate dinning

```
MAX = $max
MIN = $min

N = $n

channel pickup, putdown :union({MIN..MAX-1},{prev(MIN),next(MAX-1)})
.union({MIN..MAX-1},{prev(MIN),next(MAX-1)})
channel eat, think :union({MIN..MAX-1},{prev(MIN),next(MAX-1)})

alpha_phil(id) = {putdown.id.id,putdown.id.next(id),
pickup.id.id, pickup.id.next(id),think.id,eat.id}

alpha_fork(id) = {putdown.id.id, putdown.prev(id).id, pickup.id.id, pickup.prev(id).id}

prev(x) = if x - 1 == -1 then
N-1
else
x-1
next(x) = (x+1) % N

acquiring_release_evenst_set =
union(set(resources_order),{release_correspondent_element(e) |
e<-set(resources_order)})

print acquiring_release_evenst_set

release_correspondent_element(pickup.x.y) = putdown.x.y

acquiring_correspondent_element(putdown.x.y) = pickup.x.y

projection(P,s) = P \ diff(Events,s)
```

```
resources_order = <pickup.prev(y).y, pickup.y.y | y<-<MIN..MAX>>
print resources_order

filterSet(alpha,order) = <x | x<-order , member(x,alpha)>

hasElement(e,<>) = false
hasElement(e,<a>^order) = if e == a then
true
else
hasElement(e,order)
findElement(e,<>) = <>
findElement(e,<a>^order) = if a == e then
order
else
findElement(e,order)


ConditionUser(alpha) =
let
user_resources_order = filterSet(alpha,resources_order)
Cond(res,<a>^uro,sr) = a -> Cond(res,uro,union(sr,{release_correspondent_element(a)}))
[]
([] b : sr @ b -> Cond(res,<a>^uro,diff(sr,{b})))
Cond(res,<>,sr) = if sr != {} then
([] b : sr @ b -> Cond(res,<>,diff(sr,{b})))
else
Cond(res,res,{})
within
Cond(user_resources_order,user_resources_order,{})


ConditionResource(alpha) =
let
fork_resources_set = set(filterSet(alpha,resources_order))
Cond(res,uro,sr) = ([] a:uro @ a -> Cond(res,{},{release_correspondent_element(a)}))
[]
([] b : sr @ b -> Cond(res,res,{}))
within
Cond(fork_resources_set,fork_resources_set,{})


alphas = {alpha_phil(a), alpha_fork(a) | a <- {0..N-1}}

triple_disjoint(alpha) = card(alpha) == card({e | e <-alpha, element_count(e) <= 2})

element_count(e) = card({s| s<-alphas, member(e,s)})

Test(cond) = if cond then
SKIP
```

```
else
STOP

Fork(x) = pickup!prev(x)!x ->
putdown!prev(x)!x -> Fork(x) []
pickup!x!x -> putdown!x!x -> Fork(x)

Phil_live(y) = if y != N-1 then
think.y -> pickup.y.y -> pickup.y!next(y) ->
putdown.y.y -> putdown.y!next(y) -> eat.y -> Phil_live(y)
else
think.y -> pickup.y!next(y) -> pickup.y.y ->
putdown.y.y -> putdown.y!next(y) -> eat.y -> Phil_live(y)

Phil_dead(y) = think.y -> pickup.y.y -> pickup.y!next(y) ->
putdown.y.y -> putdown.y!next(y) -> eat.y -> Phil_dead(y)


vocab = Union({inter(a,b) | a <-alphas, b <-alphas, a != b})

print vocab
print acquiring_release_evenst_set

#if $min == 0


#end if


#for $i in range($min,$max)
assert ConditionUser(alpha_phil($i))
[T=
projection(Phil_live($i),inter(acquiring_release_evenst_set,alpha_phil($i))) #end for


#for $i in range($min,$max)
assert ConditionResource(alpha_fork($i))
[T= projection(Fork($i),inter(acquiring_release_evenst_set,alpha_fork($i)))
#end for


#for $i in range($min,$max)
assert Fork($i) :[deadlock free[F]]
assert Fork($i) :[divergence free[FD]]


#end for

#for $i in range($min,$max)
assert Phil_live($i) :[deadlock free[F]]
```

```
assert Phil_live($i) :[divergence free[FD]]


#end for#for $i in range($min,$max)
assert projection(Phil_live($i),inter(alpha_phil($i),alpha_fork($i)))
[T= projection(Fork($i),inter(alpha_phil($i),alpha_fork($i)))
assert projection(Fork($i),inter(alpha_phil($i),alpha_fork($i)))
[T= projection(Phil_live($i),inter(alpha_phil($i),alpha_fork($i)))
assert projection(Phil_live($i),inter(alpha_phil($i),alpha_fork(next($i))))
[T= projection(Fork(next($i)),inter(alpha_phil($i),alpha_fork(next($i))))
assert projection(Fork(next($i)),inter(alpha_phil($i),alpha_fork(next($i))))
[T= projection(Phil_live($i),inter(alpha_phil($i),alpha_fork(next($i))))

#end for

Forks = || a:{MIN..MAX-1} @ [alpha_fork(a)] Fork(a)

Phils = || a:{MIN..MAX-1} @ [alpha_phil(a)] Phil_live(a)


College =
Forks [Union({alpha_fork(a) | a <- {MIN..MAX-1}})
|| Union({alpha_phil(a) | a <- {MIN..MAX-1}})] Phils
```

Appendix B

Python scripts

Generate.py: Python script to generation of the dinning **philosopher configuration**

```
import os
import Config
'''
Created on Jan 19, 2012

@author: prgantonino
'''
import TemplateCSP
import sys
from TemplateCSP import Templates
from Cheetah.Template import Template
import math

def gerarArquivo(n):
    path = Config.PATH+"gerador/Gerados/MAX_"+str(n)+"/"
    if not(os.path.exists(path)):
        os.makedirs(path)

    templates = Templates(Config.PATH+"Templates/")
    name = "phils.csp"
    templateFile = templates.getPhil()
    generateFile = open(path + name, "w")
    template = Template(templateFile)
    template.max = n
    generateFile.write(str(template))
    os.chmod(path + name, 0777)

def gerarArquivoSegmentado(n):
    path = Config.PATH+"gerador/Gerados/MAX_SEG_"+str(n)+"/"
    if not(os.path.exists(path)):
        os.makedirs(path)
    ppf = Config.PROC_PER_FILE
```

```
    top = int(math.ceil(n/ppf))
    for i in range(0,top):
        templates = Templates(Config.PATH+"Templates/")
        name = "phils_div"+str(i)+".csp"
        templateFile = templates.getPhilDiv()
        generateFile = open(path + name, "w")
        template = Template(templateFile)
        if (ppf*top) > n:
            max = n
        else:
            max = (i+1)*ppf
        template.n = n
        template.max = max
        template.min = i*ppf
        generateFile.write(str(template))
        os.chmod(path + name, 0777)



if __name__ == '__main__':

    max = int(sys.argv[1])
    gerarArquivoSegmentado(max)
```

Execute.py: Python script to run the .csp files generated in FDR

```
'''
Created on Jan 30, 2012

@author: prgantonino
'''
import os
import sys
import Config
import math
import datetime

def executeAndGenerateResults(n):
    path = Config.PATH + "gerador/Gerados/Max_" + str(n) + "/"
    log_path = path + "LOG/"
    if not(os.path.exists(log_path)):
        os.makedirs(log_path)
    os.environ["FDRHOME"] = Config.FDRHOME
    os.system(
    "(time " + Config.FDRHOME + "bin/fdr2 batch " + path + "phils.csp )
    >> " + log_path + "phils.log 2>&1")
    print "Executing n="+str(n)

def checkForError(n):
```

```
    path = Config.PATH + "gerador/Gerados/Max_" + str(n) + "/"
    log_path = path + "LOG/phils.log"
    error = False
    try:
        with open(log_path, "r") as file:
            line = file.readline()
            while(line != "" and not error):
                if(line.startswith("false") or line.startswith("xfalse")
                or line.startswith(
                "The CSP compiler detected a script error or interrupt")):
                    error = True
                    print "Error script problem or assertion violated"
                line = file.readline()
        file.close()
    except:
        print "Exception"
        return 1
    return error

def executeAndGenerateResultsSeg(n):
    path = Config.PATH + "gerador/Gerados/MAX_SEG_" + str(n) + "/"
    log_path = path + "LOG/"
    if not(os.path.exists(log_path)):
        os.makedirs(log_path)
    os.environ["FDRHOME"] = Config.FDRHOME
    ppf = Config.PROC_PER_FILE
    top = int(math.ceil(n/ppf))
    for i in range(0,top):
        os.system(
        "(time " + Config.FDRHOME + "bin/fdr2 batch " + path + "phils_div"+str(i)+".csp )
        >> " + log_path + "phils_div"+str(i)+".log 2>&1")
    print "Executing n="+str(n)

def checkForErrorSeg(n):
    path = Config.PATH + "gerador/Gerados/MAX_SEG_" + str(n) + "/"
    ppf = Config.PROC_PER_FILE
    top = int(math.ceil(n/ppf))
    for i in range(0,top):
        error = False
        log_path = path + "LOG/phils_div"+str(i)+".log"
        try:
            with open(log_path, "r") as file:
                line = file.readline()
                while(line != "" and not error):
                    if(line.startswith("false") or line.startswith("xfalse") or
                    line.startswith(
                    "The CSP compiler detected a script error or interrupt")):
                        error = True
                        print "Error script problem or assertion violated"
                    line = file.readline()
```

```
            file.close()
        except:
            print "Exception"
            return 1
    return error

if __name__ == '__main__':

    max = int(sys.argv[1])
    executeAndGenerateResultsSeg(max)
```

Config.py: Configuration file for genertion

```
import os

FDRHOME = "/Users/prgantonino/Pedro/Mestrado/fdr-2.94-academic-osx/"
PATH = "/Users/prgantonino/Dropbox/Mestrado/WorkspaceMestrado/Ferramenta3.0/"
PROC_PER_FILE = 5if __name__ == '__main__':
print str(os.path.exists(PATH + "auxiliar.csp") and os.path.exists(FDRHOME + "bin/"))
```

TemplateCSP.py: Script to load Cheetah template

```
'''
Created on Jan 19, 2012

@author: prgantonino
'''

class Templates:

    def __init__(self, templatesPath):
        self.path = templatesPath

    def getTemplate(self, name):
        with open(self.path + name, "r") as f:
            dining = f.read()
        if(len(dining) > 0):
            return dining

    def getPhil(self):
        return self.getTemplate("phil.template")

    def getPhilDiv(self):
        return self.getTemplate("phil_div.template")
```

Appendix C

Example of generated file

Example of generated file with 5 philosophers and 5 forks

```
MAX = 5
MIN = 0

N = 10

channel pickup, putdown :union({MIN..MAX-1},{prev(MIN),next(MAX-1)})
.union({MIN..MAX-1},{prev(MIN),next(MAX-1)})
channel eat, think :union({MIN..MAX-1},{prev(MIN),next(MAX-1)})

alpha_phil(id) = {putdown.id.id,putdown.id.next(id),
pickup.id.id, pickup.id.next(id),think.id,eat.id}

alpha_fork(id) = {putdown.id.id, putdown.prev(id).id, pickup.id.id, pickup.prev(id).id}

prev(x) = if x - 1 == -1 then
N-1
else
x-1
next(x) = (x+1) % N

acquiring_release_evenst_set =
union(set(resources_order),{release_correspondent_element(e) |
e<-set(resources_order)})

print acquiring_release_evenst_set

release_correspondent_element(pickup.x.y) = putdown.x.y

acquiring_correspondent_element(putdown.x.y) = pickup.x.y

projection(P,s) = P \ diff(Events,s)

resources_order = <pickup.prev(y).y, pickup.y.y | y<-<MIN..MAX>>
```

```
print resources_order

filterSet(alpha,order) = <x | x<-order , member(x,alpha)>

hasElement(e,<>) = false
hasElement(e,<a>^order) = if e == a then
true
else
hasElement(e,order)
findElement(e,<>) = <>
findElement(e,<a>^order) = if a == e then
order
else
findElement(e,order)


ConditionUser(alpha) =
let
user_resources_order = filterSet(alpha,resources_order)
Cond(res,<a>^uro,sr) = a -> Cond(res,uro,union(sr,{release_correspondent_element(a)}))
[]
([] b : sr @ b -> Cond(res,<a>^uro,diff(sr,{b})))
Cond(res,<>,sr) = if sr != {} then
([] b : sr @ b -> Cond(res,<>,diff(sr,{b})))
else
Cond(res,res,{})
within
Cond(user_resources_order,user_resources_order,{})


ConditionResource(alpha) =
let
fork_resources_set = set(filterSet(alpha,resources_order))
Cond(res,uro,sr) = ([] a:uro @ a -> Cond(res,{},{release_correspondent_element(a)}))
[]
([] b : sr @ b -> Cond(res,res,{}))
within
Cond(fork_resources_set,fork_resources_set,{})


alphas = {alpha_phil(a), alpha_fork(a) | a <- {0..N-1}}

triple_disjoint(alpha) = card(alpha) == card({e | e <-alpha, element_count(e) <= 2})

element_count(e) = card({s| s<-alphas, member(e,s)})

Test(cond) = if cond then
SKIP
else
STOP
```

```
Fork(x) = pickup!prev(x)!x ->  putdown!prev(x)!x -> Fork(x)
[] pickup!x!x -> putdown!x!x -> Fork(x)

Phil_live(y) = if y != N-1 then
think.y -> pickup.y.y -> pickup.y!next(y) ->
putdown.y.y -> putdown.y!next(y) -> eat.y -> Phil_live(y)
else
think.y -> pickup.y!next(y) -> pickup.y.y ->
putdown.y.y -> putdown.y!next(y) -> eat.y -> Phil_live(y)

Phil_dead(y) = think.y -> pickup.y.y -> pickup.y!next(y) ->
putdown.y.y -> putdown.y!next(y) -> eat.y -> Phil_dead(y)


vocab = Union({inter(a,b) | a <-alphas, b <-alphas, a != b})

print vocab
print acquiring_release_evenst_set

assert ConditionUser(alpha_phil(0))
[T= projection(Phil_live(0),inter(acquiring_release_evenst_set,alpha_phil(0)))
assert ConditionUser(alpha_phil(1))
[T= projection(Phil_live(1),inter(acquiring_release_evenst_set,alpha_phil(1)))
assert ConditionUser(alpha_phil(2))
[T= projection(Phil_live(2),inter(acquiring_release_evenst_set,alpha_phil(2)))
assert ConditionUser(alpha_phil(3))
[T= projection(Phil_live(3),inter(acquiring_release_evenst_set,alpha_phil(3)))
assert ConditionUser(alpha_phil(4))
[T= projection(Phil_live(4),inter(acquiring_release_evenst_set,alpha_phil(4)))


assert ConditionResource(alpha_fork(0))
[T= projection(Fork(0),inter(acquiring_release_evenst_set,alpha_fork(0)))
assert ConditionResource(alpha_fork(1))
[T= projection(Fork(1),inter(acquiring_release_evenst_set,alpha_fork(1)))
assert ConditionResource(alpha_fork(2))
[T= projection(Fork(2),inter(acquiring_release_evenst_set,alpha_fork(2)))
assert ConditionResource(alpha_fork(3))
[T= projection(Fork(3),inter(acquiring_release_evenst_set,alpha_fork(3)))
assert ConditionResource(alpha_fork(4))
[T= projection(Fork(4),inter(acquiring_release_evenst_set,alpha_fork(4)))


assert Fork(0) :[deadlock free[F]]
assert Fork(0) :[divergence free[FD]]


assert Fork(1) :[deadlock free[F]]
assert Fork(1) :[divergence free[FD]]
```

```
assert Fork(2) :[deadlock free[F]]
assert Fork(2) :[divergence free[FD]]


assert Fork(3) :[deadlock free[F]]
assert Fork(3) :[divergence free[FD]]


assert Fork(4) :[deadlock free[F]]
assert Fork(4) :[divergence free[FD]]assert Phil_live(0) :[deadlock free[F]]
assert Phil_live(0) :[divergence free[FD]]


assert Phil_live(1) :[deadlock free[F]]
assert Phil_live(1) :[divergence free[FD]]


assert Phil_live(2) :[deadlock free[F]]
assert Phil_live(2) :[divergence free[FD]]


assert Phil_live(3) :[deadlock free[F]]
assert Phil_live(3) :[divergence free[FD]]


assert Phil_live(4) :[deadlock free[F]]
assert Phil_live(4) :[divergence free[FD]]

assert projection(Phil_live(0),inter(alpha_phil(0),alpha_fork(0)))
[T= projection(Fork(0),inter(alpha_phil(0),alpha_fork(0)))
assert projection(Fork(0),inter(alpha_phil(0),alpha_fork(0)))
[T= projection(Phil_live(0),inter(alpha_phil(0),alpha_fork(0)))
assert projection(Phil_live(0),inter(alpha_phil(0),alpha_fork(next(0))))
[T= projection(Fork(next(0)),inter(alpha_phil(0),alpha_fork(next(0))))
assert projection(Fork(next(0)),inter(alpha_phil(0),alpha_fork(next(0))))
[T= projection(Phil_live(0),inter(alpha_phil(0),alpha_fork(next(0))))

assert projection(Phil_live(1),inter(alpha_phil(1),alpha_fork(1)))
[T= projection(Fork(1),inter(alpha_phil(1),alpha_fork(1)))
assert projection(Fork(1),inter(alpha_phil(1),alpha_fork(1)))
[T= projection(Phil_live(1),inter(alpha_phil(1),alpha_fork(1)))
assert projection(Phil_live(1),inter(alpha_phil(1),alpha_fork(next(1))))
[T= projection(Fork(next(1)),inter(alpha_phil(1),alpha_fork(next(1))))
assert projection(Fork(next(1)),inter(alpha_phil(1),alpha_fork(next(1))))
[T= projection(Phil_live(1),inter(alpha_phil(1),alpha_fork(next(1))))

assert projection(Phil_live(2),inter(alpha_phil(2),alpha_fork(2)))
[T= projection(Fork(2),inter(alpha_phil(2),alpha_fork(2)))
```

```
assert projection(Fork(2),inter(alpha_phil(2),alpha_fork(2)))
[T= projection(Phil_live(2),inter(alpha_phil(2),alpha_fork(2)))
assert projection(Phil_live(2),inter(alpha_phil(2),alpha_fork(next(2))))
[T= projection(Fork(next(2)),inter(alpha_phil(2),alpha_fork(next(2))))
assert projection(Fork(next(2)),inter(alpha_phil(2),alpha_fork(next(2))))
[T= projection(Phil_live(2),inter(alpha_phil(2),alpha_fork(next(2))))

assert projection(Phil_live(3),inter(alpha_phil(3),alpha_fork(3)))
[T= projection(Fork(3),inter(alpha_phil(3),alpha_fork(3)))
assert projection(Fork(3),inter(alpha_phil(3),alpha_fork(3)))
[T= projection(Phil_live(3),inter(alpha_phil(3),alpha_fork(3)))
assert projection(Phil_live(3),inter(alpha_phil(3),alpha_fork(next(3))))
[T= projection(Fork(next(3)),inter(alpha_phil(3),alpha_fork(next(3))))
assert projection(Fork(next(3)),inter(alpha_phil(3),alpha_fork(next(3))))
[T= projection(Phil_live(3),inter(alpha_phil(3),alpha_fork(next(3))))

assert projection(Phil_live(4),inter(alpha_phil(4),alpha_fork(4)))
[T= projection(Fork(4),inter(alpha_phil(4),alpha_fork(4)))
assert projection(Fork(4),inter(alpha_phil(4),alpha_fork(4)))
[T= projection(Phil_live(4),inter(alpha_phil(4),alpha_fork(4)))
assert projection(Phil_live(4),inter(alpha_phil(4),alpha_fork(next(4))))
[T= projection(Fork(next(4)),inter(alpha_phil(4),alpha_fork(next(4))))
assert projection(Fork(next(4)),inter(alpha_phil(4),alpha_fork(next(4))))
[T= projection(Phil_live(4),inter(alpha_phil(4),alpha_fork(next(4))))


Forks = || a:{MIN..MAX-1} @ [alpha_fork(a)] Fork(a)

Phils = || a:{MIN..MAX-1} @ [alpha_phil(a)] Phil_live(a)


College = Forks [Union({alpha_fork(a) | a <- {MIN..MAX-1}})
|| Union({alpha_phil(a) | a <- {MIN..MAX-1}})] Phils
```

Appendix D

Case study example for 3 philosophers and 3 forks

```
--Resource sharing

MAX = 3

channel pickup, putdown :{0..MAX-1}.{0..MAX-1}
channel eat, think :{0..MAX-1}

alpha_phil(id) =
{putdown.id.id,putdown.id.next(id),pickup.id.id,
pickup.id.next(id),think.id,eat.id}
alpha_fork(forkid) =
let
id = forkid-MAX
within
{putdown.id.id, putdown.prev(id).id, pickup.id.id, pickup.prev(id).id}

Acquire(pidU,pidR) = pickup.pidU.(pidR-MAX)

Release(pidU,pidR) = putdown.pidU.(pidR-MAX)

Resources = {MAX..MAX+MAX-1}
Users = {0..MAX-1}

prev(x) = (x-1) % MAX
next(x) = (x+1) % MAX

print prev(1)

Fork(forkid) =
let
x = (forkid-MAX)
within
```

```
pickup!prev(x)!x ->  putdown!prev(x)!x -> Fork(forkid)
[] pickup!x!x -> putdown!x!x -> Fork(forkid)

Phil_live(y) = if y != MAX-1 then
think.y -> pickup.y.y -> pickup.y!next(y) ->
putdown.y.y -> putdown.y!next(y) -> eat.y -> Phil_live(y)
else
think.y -> pickup.y!next(y) -> pickup.y.y ->
putdown.y.y -> putdown.y!next(y) -> eat.y -> Phil_live(y)

Phil_dead(y) = think.y -> pickup.y.y -> pickup.y!next(y) ->
putdown.y.y -> putdown.y!next(y) -> eat.y -> Phil_dead(y)


Phils_set_live = { (pid,Phil_live(pid), alpha_phil(pid)) | pid<-Users}
Phils_set_dead = { (pid,Phil_dead(pid), alpha_phil(pid)) | pid<-Users}

Forks_set = { (pid,Fork(pid), alpha_fork(pid)) | pid<-Resources}

Network_alive = union(Forks_set,Phils_set_live)
Network_dead = union(Forks_set,Phils_set_dead)

ResourcesU(pid) = {pid+MAX,next(pid)+MAX}
UsersR(pid) = {pid-MAX, prev(pid-MAX)}

Resource(pidR) =
let
resource_users = UsersR(pidR)
within
[] pidU :resource_users @
Acquire(pidU,pidR) -> Release(pidU,pidR) -> Resource(pidR)

UserCondition(pidU,resources_acquired, order) =
let
 user_resources = ResourcesU(pidU)
within
([] pidR : higher(max(resources_acquired,order),order,user_resources) @
Acquire(pidU,pidR) ->
UserCondition(pidU,resources_acquired^<pidR>, order))
[]
([] pidR : set(resources_acquired) @
Release(pidU,pidR) ->
UserCondition(pidU,removeR(pidR,resources_acquired), order))

max(resource_acquired^<a>, order) = a
max(<>, order) = -1

print higher(-1, <>, ResourcesU(2))

higher(element, order, user_resources) =
```

```
{ pidR| pidR<- user_resources , pidR > element}

removeR(r,<>) = <>
removeR(r,ra^<a>) =
if a == r then
ra
else
removeR(r,ra)^<a>

print UserThatCanAquire(4,Network_alive)

higherAlreadyAquired(pidR,resources_acquired) =
{pid | pid<-resources_acquired, pidR < pid}

print higherAlreadyAquired(4,{5,6})

Pair_live(pidU,pidR) =
Fork(pidR) [alpha_fork(pidR)||alpha_phil(pidU)] Phil_live(pidU)
Pair_dead(pidU,pidR) =
Fork(pidR) [alpha_fork(pidR)||alpha_phil(pidU)] Phil_dead(pidU)

assert Fork(3) :[deadlock free [F]]
assert Fork(4) :[deadlock free [F]]
assert Fork(5) :[deadlock free [F]]

assert Phil_live(0) :[deadlock free [F]]
assert Phil_live(1) :[deadlock free [F]]
assert Phil_live(2) :[deadlock free [F]]

assert Phil_dead(0) :[deadlock free [F]]
assert Phil_dead(1) :[deadlock free [F]]
assert Phil_dead(2) :[deadlock free [F]]

assert Pair_live(0,3) :[deadlock free]
assert Pair_live(0,4) :[deadlock free]
assert Pair_live(1,4) :[deadlock free]
assert Pair_live(1,5) :[deadlock free]
assert Pair_live(2,5) :[deadlock free]
assert Pair_live(2,3) :[deadlock free]

assert Pair_dead(0,3) :[deadlock free]
assert Pair_dead(0,4) :[deadlock free]
assert Pair_dead(1,4) :[deadlock free]
assert Pair_dead(1,5) :[deadlock free]
assert Pair_dead(2,5) :[deadlock free]
assert Pair_dead(2,3) :[deadlock free]

assert Fork(3) [T= Resource(3)
assert Resource(3) [T= Fork(3)
```

```
assert Fork(4) [T= Resource(4)
assert Resource(4) [T= Fork(4)

assert Fork(5) [T= Resource(5)
assert Resource(5) [T= Fork(5)

AcquireU(pidU) = {Acquire(pidU,pidR) | pidR<-ResourcesU(pidU)}
ReleaseU(pidU) = {Release(pidU,pidR) | pidR<-ResourcesU(pidU)}

assert UserCondition(0,<>, <>) [T= Projection_phil_live(0)
assert UserCondition(1,<>, <>) [T= Projection_phil_live(1)
assert UserCondition(2,<>, <>) [T= Projection_phil_live(2)

Projection_phil_live(pid) =
(Phil_live(pid) \ diff(Events,union(AcquireU(pid),ReleaseU(pid))))
Projection_phil_dead(pid) =
(Phil_dead(pid) \ diff(Events,union(AcquireU(pid),ReleaseU(pid))))

assert UserCondition(0,<>, <>) [T= Projection_phil_dead(0)
assert UserCondition(1,<>, <>) [T= Projection_phil_dead(1)
assert UserCondition(2,<>, <>) [T= Projection_phil_dead(2)



Behaviour(N) = || (pid,P,A) : N @ [A] P

assert Behaviour(Network_alive) :[ deadlock free [F]]

assert Behaviour(Network_dead) :[ deadlock free [F]]
```