Concurrent program metrics drawn by QUASAR numbers

Claude Kaiser, Christophe Pajault, Jean-François Pradat-Peyre
CEDRIC - CNAM Paris
292, rue St Martin, 75003 Paris
{kaiser, pajault, peyre}[at]cnam[dot]fr
fax : 01 40 27 28 45
http://quasar.cnam.fr/

Abstract. Aiming at developing reliable concurrent software, the engineering practice uses appropriate metrics. Quasar, the tool that we develop for automatically analyzing the concurrent part of programs, provides some data as a result of its program analysis. We attempt to use them as metrics and to show their usefulness for software engineering and for reliable programming control. In addition to the validation of code, some delivered data may be relevant for marking the quality of code; we show that they may be useful for comparing different concurrency semantics, for comparing the execution indeterminism of different implementations of a concurrency pattern and for estimating the scalability of a solution. As a case study for providing data and insights, we present and analyze with Quasar several implementations of a distributed symmetric non-deterministic rendezvous algorithm. We consider also two possible uses of these data for indeterminacy estimation and for concurrent software quality. We conclude with a feedback on software practice.

Keywords. Concurrent software metrics. Distributed symmetrical non-deterministic rendezvous. Concurrency indeterminism. Concurrency analysis and metrics.

1. Concurrent program metrics

Concurrent program metrics is one among reliable software technologies. Reliability and performance of concurrent software concern on the one hand the program functionalities (and the relevance of the values computed by the program) and on the other hand the program concurrent behaviour. The asynchronous execution of concurrent processes is highly non deterministic. Thus concurrency introduces temporal dimensions of correctness, i.e., safety and liveness. Safety properties of concurrent programs concern the absence of deadlock or of livelock, the coherence of shared variable and the absence of race conditions, the respect of mutual exclusion when using some shared resource. Liveness properties concern the absence of starvation, the guarantee that some service will eventually be done for a given client. Thus they are central concerns in any concurrent design and its implementation.

Metrics are used for program quality control. Following the seminal works of McCabe in 1976 [29] and Halstead in 1977 [21], several metrics have been developed for program quality control (McCabe [30], Chidamber [11], Brito [5], Bensiya [4], Spinellis [35],…] and are now available through specific software engineering tools such as Telelogic Logiscope, IBM-Rational tools suite, IPL klockwork, …

Additional metrics may help explaining, analyzing, and providing a thorough understanding of the possible behaviours resulting from the inherent indeterminism of asynchronous concurrency in a concurrent program.

Let us examine some metrics, which are suitable for concurrent programs.

1.1. Algorithmic capability

Prior to any implementation, while selecting a concurrent algorithm, its algorithmic complexity allows choosing among different safe solutions. Usual aspects of this algorithmic capability include:

- Efficiency in computing resource use and in data traversal or retrieval,

- Parallelism ratio: i.e. effective computer parallelism usage when precedence relations constraints restrain the concurrency or effective usage shared resources (for example memory), when deadlock avoidance leads to postpone their allocation [24],

- Response time in real time applications [8, 14]

- Impact of concurrency liveness requirements on efficiency: for example solutions tolerating starvation with some low probability may use resources more efficiently than when starvation is strictly forbidden; priority given to some clients may be detrimental to fairness.

1.2. Quality of solution programming (software engineering practice)

The way a program is written can have some important consequences on software development productivity, program testing, maintenance and evolutivity, even on execution supervision (allowing to observe and schedule possible behaviours). Many source code programming style guides (such as GSS [20], JavaRanch [23]) stress qualitative characteristics such as style, readability, conciseness, good structuration, absence of design and coding errors. Metrics for program text quality try to quantify some characteristics such as size of lines of code, complexity (cyclomatic number measuring the number of linearly independent paths through a program's source code), and specific design features.

Concurrent programming introduces new complications since concurrent programs can have many instances of execution for the same set of input data. The quality metrics must then include at least the number and characteristics of tasks and of intertasks interactions [12, 34].

1.3. Observability of execution behaviour

The observability and the understanding of application programs are requested when monitoring, debugging, reexecuting or checkpointing them. For concurrent programs this is complicated by the amount of intertasking activity, nondeterminism and data access concurrency. Having an idea of the amount of indeterminism due to the asynchronous concurrency is useful when analyzing the current state of the application displayed on a synoptic control panel.

A metrics based on information communication theory [10] has been proposed. An execution of a concurrent program is interpreted as an experiment that selects, from all the possible ways that a program allows, the single correct configuration for that execution. The information content, the amount of intertasking activity and the nondeterminism contained in this concurrent program represented on a corresponding concurrency graph, contributes to this measure. Unfortunately the number of states in the corresponding concurrency graph can be very large, thereby limiting the programs that can be measured. To cope with this limitation, another metrics based on the entropy function of communication theory, i.e., the number of bits required to encode the program, is used as a measure of uncertainty.

1.4. Comparing concurrent programs behaviour

A concurrent problem can lead to several approaches. Given a paradigm or a pattern, there exist different algorithmic solutions [8, 28]. On the contrary to sequential programs, concurrency programming cannot ignore the behaviour of the underlying platform which implements processes and provides data sharing and message passing for process synchronization and cooperation. Given an algorithmic solution there exist different implementations, resulting from the choice of a language, from programming style and from the concurrency run-time platform.

A taxonomy is needed for comparing the different possible solutions for a concurrency paradigm and the different implementations of a chosen solution. A metrics is therefore needed for elaborating this taxonomy.

1.5. Scalability

The size of applications is determined mainly by the number of concurrent entities, mapped in concurrent processes. A metrics should allow giving the influence of scale to concurrency complexity according to the number of cooperating processes.

2. Quasar

Quasar is a static analysis tool based on the use of slicing [36, 37] and of model checking [13]. It output data, which we attempt to use as concurrency metrics. We present below its structure and the data delivered.

2.1. Quasar structure

Quasar provides software engineers with an automatic concurrency analysis tool. It takes as input the application source code and uses it for generating and validating a semantic model.

As any validation method, this one has to face the explosion of the number of states which need to be analyzed. In order to be able to analyze the concurrent behaviour of real-life programs, the number of states that Quasar must consider is automatically reduced as soon as possible and as much as the program structure allows it.

The analysis of the program concurrent behaviour is performed in four automatic steps.

1. First, the original text of the program is automatically sliced in order to remove those parts of the program that are not relevant to the concurrency property the user wants to check. This allows generating a model which is smaller than the one corresponding to the whole program, but which has the same behaviour according to the investigated concurrency property. The sliced program is an executable program. Usually the slicing removes a lot of data which otherwise would uselessly obscure model checking. This removes for example features which would lead to an infinite model. This smaller program will be easier to analyze in the next steps.

2. Second, the sliced program is translated into a concurrency model using a library of patterns. A pattern is a meta-net corresponding to a declaration, a statement or an expression. Each pattern definition is recursive in the way that its definition may contain one or several others patterns. The target model mapping the whole sliced program is obtained by replacing each meta-net by its corresponding concrete sub-nets and by merging all sub-nets. This mapping is optimized by performing structural reductions on the target model. In particular these structural reductions, also called static reductions, merge causally connected transitions using generalized structural agglomeration techniques. This optimization smoothes also the coarseness of the automatic translation.

3. In the third step Quasar checks the required property on the target model, using graph reduction and structural techniques (delta marking allowing a symbolic storage of states, coloured stubborn sets technique) for optimizing state based enumeration (model-checking). These reductions are called dynamic reductions.

4. At last, if the required property is not verified, the state in which the application is faulty and a report demonstrating a sequence invalidating the property are displayed.

A detailed description of this process can be found in [2, 15, 32].

A preliminary version has shown the feasibility of the approach [6]. A second version allowed experimenting several advanced aspects (as analyzing shared memory use without mutual exclusion locks [16], as adding temporal logic for safety properties, as dynamic task creation and dynamic creation of objects [17]).

The present version has been written for introducing new techniques and state representations and thus is optimized for limiting the state explosion during program analysis [19, 33].

2.2. Quasar logs

The current Quasar version (http://quasar.cnam.fr/) analyzes an Ada concurrent program, translates its sliced version into a high level (coloured) Petri net and compiles this net into a reachability graph which is traversed for model checking.

The first result provided by Quasar is to indicate whether the required property holds (for example absence of deadlock). If not it displays a running sequence invalidating the property.

Quasar displays the sliced executable program. This allows using it for program inspection.

Quasar second step ends generating a coloured Petri net model of the sliced program. The number of places and of transitions is output.

In the third step, Quasar performs model checking, generating a reachability graph which records all possible different executions of the program. This step is also optimized. Different items of this step are recorded, especially the number of graph states, the number of graph arcs and the number of arcs visited during the graph traversal, the graph compilation time and the graph searching time for model checking, the size of the memory used for storing the graph elements.

The memory size and CPU time of a Quasar execution are also available.

3. Comparing several versions of a concurrent non-deterministic pairing

Non-deterministic pairing has been chosen as a case study. Several implementations are available and their static analysis provides data, which serve as a basis for their comparison and for a metrics definition attempt.

3.1. Non-deterministic symmetric pairing

Non-deterministic pairing occurs in a system when a concurrent process becomes candidate for constituting a pair with any other candidate process. The pairing is said symmetrical since candidate partners behave all similarly, i.e. they have all the same capabilities for sending or receiving partners requests. The pair is the result of the non-deterministic interaction between two (or more) candidate partners. Once paired the processes are no longer candidates and become partners. One of them is chosen for leading the pair interactions. The leader role is to avoid calling deadlock by introducing some dissymmetry: the leader calls while the other accepts a call. The partnership ends after a while allowing both processes of the pair to return to the state of possible candidate partners. The absence of candidate partners will not last forever.

The functionality required is very simple; once two candidates A and B have been selected for being connected, the pairing specification is:

{(A, Partner(A) = nil, Leader(A) = nil) and (B, Partner(B) = nil, Leader(B) = nil)}
Pairing
{(A, Partner(A) = B, Leader(A) = Leader(B)) and (B, Partner(B) = A, Leader(B) = Leader(A))}

The challenge is to carry out a connexion as soon as possible once two processes at least are candidates. Pairing must concern two and only two processes. No third process should be able to disturb the creation of a unique pair and the notification to each pair member of its partner name.

We consider a distributed system made of a set of at least two asynchronous non-failing concurrent processes and we propose to examine two algorithms. In the first one, a candidate process takes advantage of a shared pairing service; in the second one, it has to send requests to other processes until it finds one which is also candidate. A more detailed specification, which is not in the scope of this paper, can be found in [25, 26].

In this paper, we focus on several implementations of this pairing, considered as a concurrency pattern.

3.2. The different implementations of anonymous non-deterministic pairing

We experiment several solutions with different implementations of the pairing component corresponding to various concurrency features. The main program declares and creates N concurrent processes and calls each of them for allocating unique Ids. Thus each process starts accepting a call for grasping its unique Id and then loops forever. Each cycle starts by a call to the pairing component for getting a current partner and performing a peer-to-peer transaction with it. This partnership is delimited by two rendezvous: Start_Peering and Finish_Peering.

3.2.1. Shared agora

In this first solution for the shared service, a meeting place called an Agora is known and used by all seeking processes as a pairing data container [25]. Implementations experimented use shared data and shared procedures controlled by semaphores or by a monitor [22].

The semaphore implementation (*agoraserverposix*) programs explicitly a mutual exclusion and uses the synchronization technique called passing the baton [3]. It corresponds to the style of concurrency expression allowed by POSIX. In our experimentation, each semaphore is simulated by an Ada protected object.

Several possible monitor concurrency semantics have been used in the past and a classification is presented in [7]. Every implementation provides mutual exclusion during the execution of a distinguished sequence (synchronized method in Java, lock in C#, protected object subprograms in Ada) using a lock for every object. The semantics differ in the chosen policies for blocking, signalling and awaking processes. Three kinds of monitor structure are experienced in this paper.

Three implementations use explicit self-blocking and signalling instructions and are based on Java style using “wait()”,“notify()” and “notifyAll()” clauses with a unique waiting queue per encapsulated object (termed “synchronized”). In our experimentation, the Java concurrency structure is simulated in Ada [18]. The first implementation (*agoraserverjava*) emulates a native Java concurrency semantics with its weak fairness which moves awaken threads into a unique system ready threads queue. This weak semantics requires using defensive programming in order to avoid deadlock. The other two implementations (*agoraserverjavastrong and agoraserverjavastrongdc*) enforce a strong fairness semantics and give precedence to the awaken threads over other ready threads. In the first one the defensive code has been skipped while it has been kept in the second although it is useless.

The last monitor structure implemented (*agoraserverada*) uses implicit self-blocking and signalling [27] as provided by Ada protected objects [1] with entry barriers and automatic entry re-evaluation. The strong fairness semantics results from the so-called Ada “eggshell model” for protected objects which gives precedence to awaken calls over new calls. A “requeue” statement to a private entry enables the first request to be postponed.

3.2.2. Shared remote server

In this second solution for the shared pairing service, an additional process is used by all seeking processes as a shared pairing server. Two implementations (*rendezvousservermonitor* and *rendezvousservernested*) take advantage of the Ada rendezvous between tasks. The task of the first one acts as a monitor structure for its clients while the task of the second one realizes an anonymous rendezvous nesting two calling client tasks [26].

3.2.3. Distributed cooperation

The third solution [26] is fully distributed and does not use a common agora or a common server. Each candidate process consults other processes until it meets a candidate process and, since the pairing is symmetrical and non-deterministic, it must also answer requests of other candidate processes. This behaviour excludes a deadlock situation where all processes have called the others or where all processes wait for a call. Thus each process must be non-deterministically either requesting or listening. A requesting candidate sends only one request at a time (there is no calling concurrency to manage). A listening process picks and serves its received requests one at a time (there is no listening concurrency to manage). Each called processes must answer indicating whether it is candidate or not. The pairing implies that both partners are candidates and that one is requesting and the other is listening. Finally the pairing decision is taken by the listening partner (thus no distributed consensus is required).

There is only one implementation (*cooperativeadatask*); it requires associating each process with a local assistant task, which takes advantage of the Ada selective non-deterministic rendezvous between tasks and which cooperates with other assistant tasks.

3.2.4. Dummy component

The aim of this implementation (*dummy*) is to provide a low bound model with only a shared procedure call which returns a predefined partner which is the same for all candidates (since this call is unuseful for concurrency, it is normally sliced. However Quasar provides a mean for signalling that a statement should not be sliced). All transactions are performed with the fixed predefined partner and are no longer peer-to-peer transactions. This choice preserves a reliable solution.

3.2.5. List of experimented implementations

Monitor like implementations:
agoraserverada : Ada protected object : reliable
agoraserverjava : native Java semantics: reliable since defensive programming is introduced
agoraserverjavastrong : Java with a modified semantics providing strong fairness
agoraserverposix ; Semaphore solution programming explicitly a monitor like structure

Remote server implementations:
rendezvousservermonitor: Ada style for implementing a monitor with a server task : reliable
rendezvousservernested: Ada rendezvous server task : reliable

Distributed cooperation:
cooperativeadatask: fully distributed with the addition of a local assistant task : reliable

Dummy component:
dummy: a fixed partner is member of every pair : reliable

3.3. The verdict returned by Quasar

All these implementations have been ultimately written in Ada and use Ada tasks as process structure, declaration and activation. When POSIX semaphores and Java monitors are concerned, they have been emulated and their emulation is used instead of Ada concurrency features directly. All implementations have thus been able to be analyzed by Quasar.

The first objective of Quasar is to indicate whether a given concurrency property holds or not. All the implementations from which data have been collected have been proven deadlock free by Quasar. Beforehand some, which had been detected as faulty, have been analyzed with the report provided by Quasar and corrected by adding defensive code (*agoraserverjava* and *rendezvousservermonitor*).

The data collected by Quasar running on a uniprocessor are given in Annex in different tables.

The first group of data concerns the coloured Petri net model. The sliced Ada program, which contains only concurrency relevant text, is automatically compiled into its coloured Petri net model. This model is simplified by structural reductions, which operate on superfluous places or transitions due to verbose programming or to the automatic generation in the compilation phase. #Places/Reduced (resp. #Transitions/Reduced) records the number of places (resp. transitions) in the original and in the reduced Petri net model.

The second group of data concerns the reachability graph that is built during the model-checking step. It records the number of states and the number of arcs visited for checking that the required property is not violated. The ratio compares the sizes of the graphs for two successive values of the number of concurrent processes.

The last group of data concerns CPU and memory used by Quasar for its analysis. Execution records the time (in seconds) needed for compiling the Petri net and its reachability graph and the time passed in the graph traversal while model checking.

3.4. Metrics and insights derived from the collected data

We try now to derive some insight from the data recorded for the different implementations of the pairing component. The results are summarized Figure 1 and fully recorded Table 1.

3.4.1. Quality of code

The size of the Petri net and the amount of static reductions give a flavour of the conciseness of code and of the style of programming. It can be used as a quality assessment metrics for the concurrent part of a program, extending the cyclomatic complexity of functions. Note that the coloured Petri net of a given implementation is the same for all value N of the number of concurrent processes. Note also that Quasar performs a static reduction of about 4.

As foreseeable the distributed cooperation is more complex than the shared service. Surprisingly the reduced Petri net of the distributed solution is only twice as large as the rendezvous server monitor which uses also rendezvous between tasks.

All shared paring service solutions are roughly equivalent. This reflects the fact that they use the same algorithm. The differences result from the additional code used for emulation. For POSIX three semaphores are used each being emulated by a protected object. The pure Ada solution needs only one protected object. For Java, only one protected object is used for emulating the Java object, but moving a signalled task to the ready queue and forcing it to call the method in a loop (respecting the Java style) must be explicitly programmed since this contravenes the regular Ada protected object policy ("the eggshell model").

The equanimity of the metrics reflects the fact that all the examples have been programmed with the same algorithm and by the same person(s). It would be interesting to compare implementations made by differently skilled persons or by students and without a prescribed algorithm.

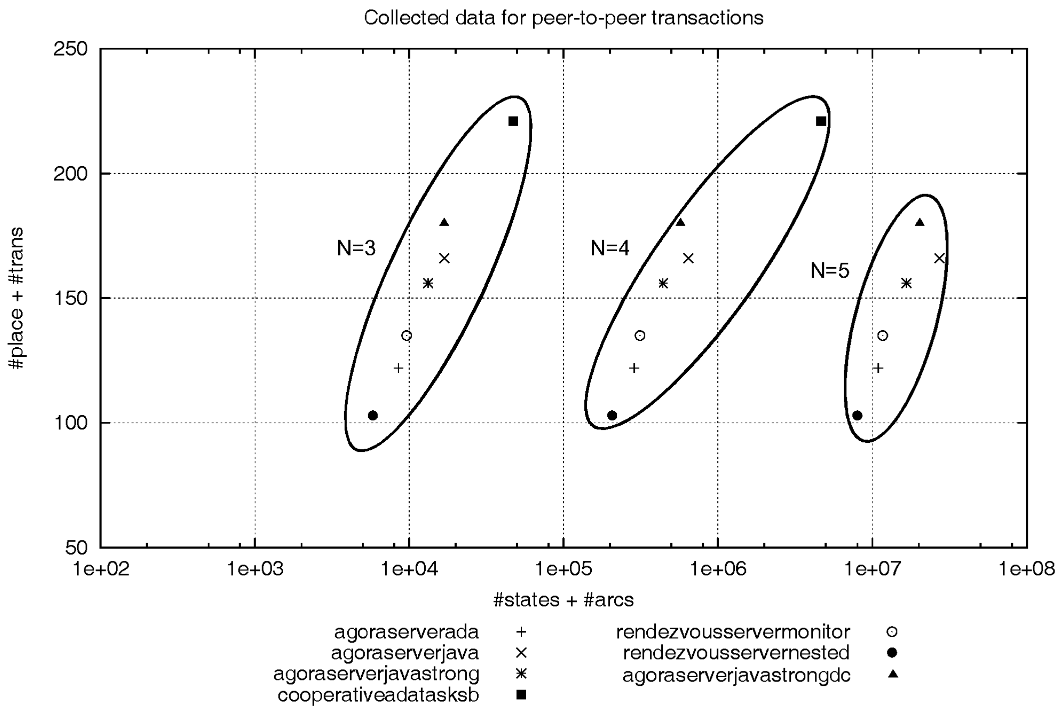

Figure 1. Graph summarizing the data collected by Quasar for peer-to-peer transaction

3.4.2. Nondeterminism

The reachability graph records all the successive states that a program execution will visit and it does it for all possible different executions of a program. Reading the data may lead to two sorts of deduction. The least number of items in the graph (states and arcs), the less complex is the program execution; but also: the least number of items, the least task interleaving. The graph size is thus related both to the execution efficiency and to the execution indeterminacy.

Again the distributed cooperation is more complex than the shared service. Let compare the distributed cooperation and the rendezvous server monitor and examine the ratio of the number of arcs visited in the reachability graph; this ratio is about 6 for 3 processes and is about 14 for 4 processes.

The shared pairing service solutions show a difference between monitor like features (in Ada and Java) and low level semaphore programming. This latter is more complex because the analysis (and the execution) cannot consider that the critical sections of code embedded by programmed P and V operations of a mutual exclusion semaphore are serializable, as it can be supposed when a monitor is used, since any other P or V operations can be programmed by error. Thus the analysis has to consider the interleaving of the instructions of the critical section instead of considering then as serializable.

The genuine Java like solution has more execution indeterminacy since the weak fairness semantics creates more process switching possibilities. When using strong fairness semantics as in *agoraserverjavastrong*, the indeterminacy is at least 25% less. If the defensive code is kept as in *agoraserverjavastrongdc*, the Petri net size is a little larger, however the indeterminacy is still less than with the regular Java semantics.

3.4.3. Comparison of implementations

The shared pairing service solutions can be ordered in terms of Petri net or reachability graph sizes. The same order holds for the Petri net size as for the reachability graph size.

For high-level concurrency structures the best is the *rendezvousservernested implementation*, since it has no shared variables and that the information is passed from input parameter to output parameter. The second best is the Ada protected object implementation, showing the quality of this feature. The increasing order is: *rendezvousservernested --> agoraserverada --> rendezvousservermonitor --> agoraserverjavastrong --> agoraserverjava --> cooperativeadatask* .

Note that being able to compare the different implementations of a concurrency pattern may be a useful criterion of choice in automatic code generation.

3.4.4. Scalability

Analyzing solutions with different process number provides some insight of the indeterminacy scalability.

For the shared pairing services, the scale factor is about 35 when they are implemented with high-level concurrency structures. It may be conjectured about 180 for the semaphore implementation. This results probably from the interleaving of the instructions within the critical section.

For the distributed cooperation, the scale factor seems to be near 100.

A distributed version of Quasar is being developed [31] and we hope that it will allow analyzing implementations with larger values of N and refine our knowledge of these scale factors

3.4.5. Comparing with a simpler utilization of the shared component

As Quasar analyzes the global behaviour of concurrent programs, not the behaviour of a component per se, we wanted to check whether the comparison of different implementations is modified when the component is used in another concurrent program. Thus another utilization of the pairing component has been programmed in a simpler program where the processes call the pairing component only and have no other interactions. The results are summarized Figure 2 and fully recorded Table 2. The preceding comparisons remain true. However some anomaly may be observed.

3.4.6. Some indeterminacy anomalies

Since the program with an empty sequence of process interaction (Quasar results recorded Table 2) is shorter than the program with a peer-to-peer transaction, which has two additional rendezvous (Quasar results recorded Table 1), its Petri net and its reachability graph should be smaller. This is not the case for all of the component implementations.

Let us consider the sets of data for the Java like code (*xs_agoraserverjava***),** for the POSIX like code **(***xs_agoraserverposix*) and for the cooperative protocol (*xs_cooperativeadatasks*). The Petri nets size is reduced as we expect it. However the reachability graph sizes are larger.

Let us examine some possible explanations.

In genuine Java, the weak fairness semantics induces much process context switching due to the move of released processes into the system ready queue. With less simultaneously possible ready processes (due to the peer to peer transaction), the size of the process ready queue is smaller. Note that this anomaly is not present when using strong fairness with Java.

For the cooperativeadatasks, the same effect occurs: there are fewer processes that are in effective possible cooperation since at least two of them are already in peer-to-peer interaction.

For agoraserverposix the mutual exclusion is not known by Quasar, thus the critical section of code is not considered as a unique serializable flow of program and all its instructions may be interleaved causing very large combinatorics.

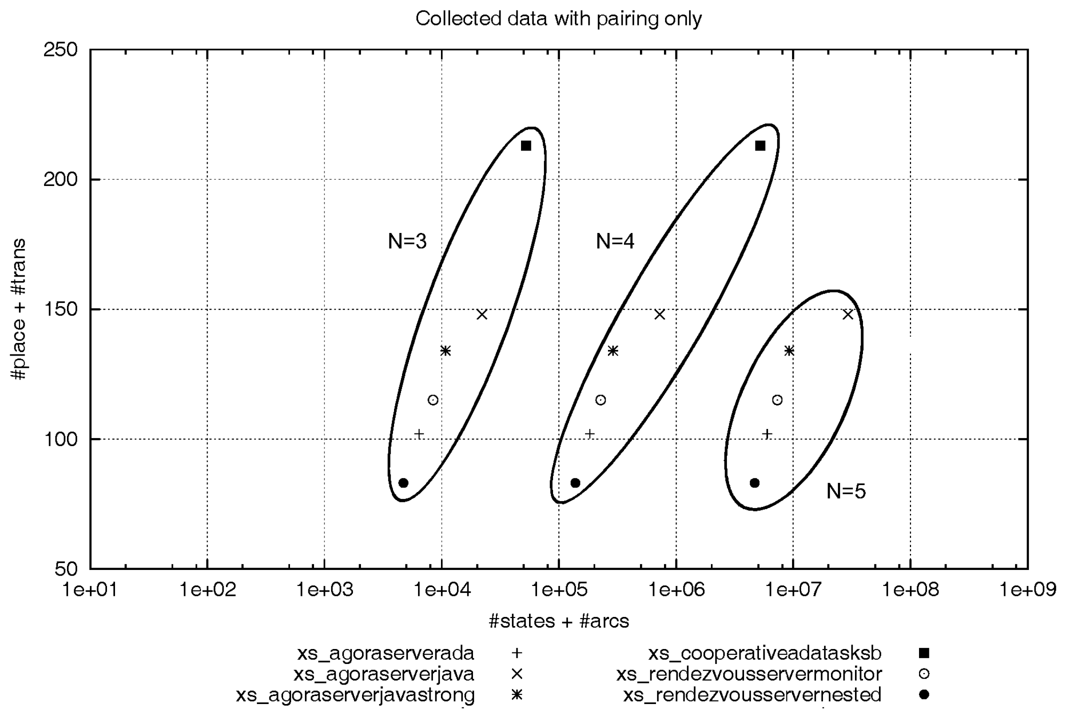

Figure 2. Graph summarizing the data collected by Quasar for pairing only

4. About indeterminacy estimation

It has been shown above that the data provided by Quasar allow comparing several implementations of a concurrent programs. We consider now two other possible uses of these data. First we observe how they evolve when a component is combined with other concurrency features. Second we examine a possible estimation of software quality.

4.1. Program code incrementations

Let us suppose that a concurrent program uses an available component, here the pairing component, jointly with other concurrency features. A basic program is being incremented, adding at each incrementation only one concurrency feature: its concurrent processes increment their interaction either by sharing an additional protected object or by having additional peer-to-peer rendezvous. This simple kind of incrementation does not complicate much the relationship between the pairing component and the processes.

As the Petri net model is built by parts, adding a pattern for each part, we may expect that the original and reduced Petri nets sizes reflect this additive structure. We inspect the data to detect whether the indeterminacy variation may be somehow predictable, although model checking evolution is not linear, even in such a simple case, and as Quasar tries to reduce at most the size of the reachability graph using the peculiar program properties.

4.1.1. The set of program code extensions

Slightly different programs that call either a dummy component or the Ada protected object component (*agoraserverada*) are used for providing insights:

xx_context: this is the program with an empty sequence of interactions which is used in section 3.4.5 and which results have already been reported Table 1.

s1_context: this program adds a protected object to the former. All cooperating processes call a parameterless shared protected procedure.

s2_context: this program adds a rendezvous to the program of xx_context. All cooperating processes use it in a peer-to-peer interaction.

s3_context: this program adds a protected object and a rendezvous to the program of xx_context. All cooperating processes perform a peer-to-peer interaction followed by a parameterless shared protected procedure call.

original_context: this is the program originally used in section 3 for comparing the component implementations. Peer to peer transaction between the cooperating processes is expressed by two calls (start and finish) between the peers. Results have been already displayed table 2
t1_context: this program adds a protected object to the former program. The peer-to-peer transaction is followed by a parameterless shared protected procedure call.
t2_context: this program adds a rendezvous to the original context. The cooperating processes perform a peer-to-peer large transaction expressed by three calls between the peers.

The set of data collected by Quasar are displayed in Tables 3, 4 and 5. The latter presents the results when Quasar is used without setting the usual static or dynamic reductions.

4.1.2. Adding a protected object in the processes code

The comparison of data variations collected in these programs for N = 3 when adding a protected object is given Table 3.

There is an almost fixed Petri net extension: about 24 places and 22 transitions in the original net, about 7 places and 4 transitions in the reduced net. The non reduced reachability graph extension shows in both cases an extension due to an additional indeterminism: tasks are either "before" the call to the protected object or "after" the call. The size of the ultimately reduced reachability graph does not necessarily follow this augmentation, thanks to reductions performed during its building.

4.1.3. Adding a rendezvous in the processes code

The comparison of data variations collected in these programs for N = 3 when adding a rendezvous is reported Table 4.

There is here also some fixed Petri net extension with again two situations: the first rendezvous introduction (in *s2_context* and in *s3_context*) adds 37 places and 36 transitions in the original net, 8 places and 7 transitions in the reduced net, while the other extensions add only 20 places and 19 transitions in the original net, 4 places and 3 transitions in the reduced net. This difference can be explained by the fact that in the last cases some places are shared between different rendezvous patterns while this is not possible in the first case (which contains a unique rendezvous).

4.1.4. About metrics composition and prediction

Evaluating the complexity of a program through the size of the corresponding Petri net or through the size of the reachability graph of the model leads to some difficulties.

At first, the Petri net size cannot be easily predicted since the Petri net is not generated by a straightforward juxtaposition of patterns; in fact the patterns are meta models of nets from which concrete subnets are derived recursively and some of these subnets places are merged to provide the final net. Thus predicting a Petri net size is as imprecise as predicting the size of the code generated by a compiler.

Second, model checking is highly combinatorial and is not a process decomposable in separate parts. Thus the number of states, and therefore the indeterminacy, grows exponentially. Consider for instance N complete independent processes each executing K instructions. The size of the reachability graph would be at least of K^N. However, we cannot consider this program as "concurrency complex" since no synchronization occurs! The xx_dummy _3 program is close to this situation, since the main program calls the 3 processes which have then no other interaction than calling a shared procedure. Its unreduced Petri net has 168 places and 157 transitions and its reachability graph without reduction has over 9 million states.

In conclusion, estimating the metrics of a concurrent program from the metrics defined for its components, as can be done for sequential code, is still an open problem.

4.2. The reduction ability as a quality factor

Quasar can be used without setting the static or dynamic reductions. This allows another examination of a concurrent program, although limited because of the huge size of the graph without reductions. Let us do this analysis for the preceding programs. The results are given Table 5 and Figure 3. They display high reduction factors, especially when the code contains few process interactions and is almost parallel. The reductions operating on the shared procedure component produce a ratio of at least 690 for static reduction, and of at least 118 108 for static followed by dynamic reduction. The respective ratios for the protected object component are 188 and 5 137. Let us explain how Quasar proceeds.

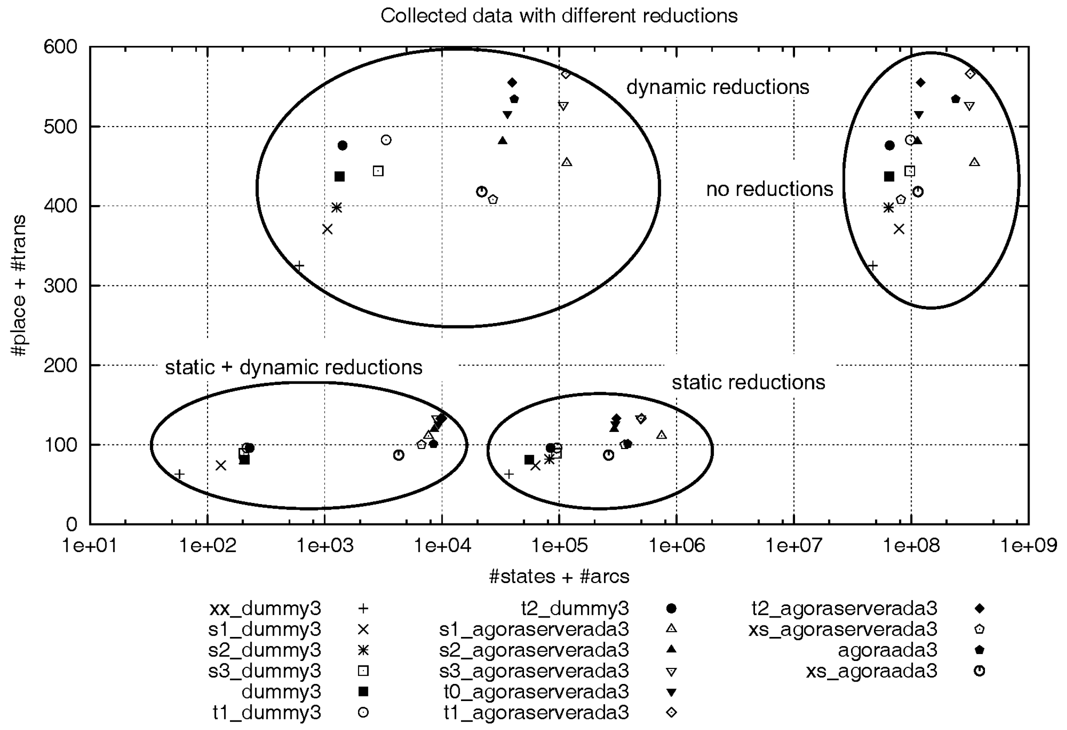

Figure 3. Graph summarizing the different reductions performed by Quasar

When processes code include sequences of statements which are serializable (in this case their behaviours are easily understandable), Quasar reduces the size of the Petri net model since these serializable sequences of actions are detected and replaced by equivalent atomic actions. We can observe this on data provided Table 5: the simpler are the sequences of actions performed by tasks, the higher is the structural (static) reduction ratio.

When performing dynamic reduction (reductions performed during the building of the reachability graph), Quasar tries to detect independent behaviours and, depending on some conditions, reduces the number of state and arcs that have to be built and explored (without altering the true value of the analysed property); thus the more independent the processes are (and consequently the more understandable is the program) the higher the dynamic reduction ratio is. This is the case for the dummy component. Recall that its shared procedure has no effect on concurrency and was prevented to be sliced. Thus it is withdrawn by any reduction. In *xx_dummy* (no additional rendezvous, no protected object) the remaining code is parallel and is analysed with 29 states only (with 9 million states without reduction, this corresponds to a reduction factor over 300 000).

Many reductions performed by Quasar are favoured by the presence in the code of purely parallel structure and symmetries, by high-level synchronisation structures (for example the monitor concept), by structuration (for example paradigm and pattern use), evidence and clarity of code. These code properties, which are required engineering practices when reliable code is needed, have also a positive side effect on concurrency analysis.

Hence, the reduction ratio seems to give some useful indication of the "concurrency complexity". Indeed all reductions performed by Quasar aim to concentrate the analysis to the genuine part of synchronization which is involved in the property examined. This allows us to claim that the ability of a program to be strongly reduced by Quasar may be considered as a credible mark of good software engineering quality. Aiming at providing such an insight we are adding, in the Quasar output data, an indication of the amount of dynamic reduction performed by Quasar when it analyzes a concurrent program.

5. Conclusion

Summary

In this paper, we have compared several implementations of a concurrent non-deterministic rendezvous algorithm. Data provided by their static analysis with Quasar have allowed:

- Verifying whether a concurrency required property holds, for example absence of deadlock,
- Giving insights on the quality of the concurrent code,
- Providing a metrics for execution indeterminacy,
- Comparing different implementations of a given concurrent pattern,
- Giving an idea of the scalability evolution of a given implementation.

We have also considered computing the indeterminism of a compound program and using the reduction factor achieved by Quasar as a concurrency quality indicator.

All the programs and the results of their Quasar analysis are available on http://quasar.cnam.fr/files/concurrency_papers.html. In a previous paper [18] we have also compared data obtained by Quasar analysis with efficiency results measured when running the analysed program.

Possible extensions

Extensions that can be envisioned include using programmer's assumptions and associating Quasar use with run time test strategies.

Programmer annotations could be used for giving assumptions of critical section of statements or of serialization of statements (similar to writer sequence and reader sequence). Quasar may take benefit of these assumptions and could be used to verify also whether the assumed properties hold or not: never more than one access to an assumed critical section, no write in an assumed serialized section of statements.

The slicing phase may be extended to generate automatically causality time stamps and to use them at run-time for performing dynamic testing as in [38] as a complement to the code analysis performed by Quasar.

Feedback on software practice for reliable concurrent programs

From this analysis we can feedback some design recommendations and design practice for better, simpler, easier to understand and to observe, i.e. more reliable, implementations.

A high level language derives benefit from a large grain of mutual exclusion access. Less indeterminacy is shown when a high level language guaranties a large granularity of concurrent access in mutual exclusion. It is then better using a monitor structure than programming explicitly a critical section with semaphores.

A monitor structure controlled by assertions improves determinism. Blocking and resuming concurrent threads with wait and signal clauses lead to more indeterminism and less observability than doing it with assertions.

Strong fairness semantics is favourable for reliability. Using wait and signal clauses with a weak fairness semantics produce more process switching than a strong fairness (as seen when comparing Java and Javastrong implementations of a monitor). Moreover weak fairness requires defensive programming to counterbalance this weakness and to provide reliability.

Using concurrency paradigms rather than ad hoc concurrency code provides code where the cooperation is easier to understand and to observe. Generally the application will show less irreducible indeterminism.

6. Acknowledgments

The team that contributed to the present version of Quasar was composed of Olivier Alzeari, Sami Evangelista, Claude Kaiser, Christophe Pajault, Jean-François Pradat-Peyre and Pierre Rousseau. (http://quasar.cnam.fr/)

7. References

1.[Ada 2001] *Consolidated Ada Reference Manual*, International Standard ANSI/ISO/IEC-8652/1995(E*)*, LNCS 2219 Springer January 2001. (ISBN 0302-9743)
2.[Alzéari 2008] Olivier Alzéari, *Quasar Observer*, mémoire d'ingénieur CNAM, 2007
3.[Andrews 1991] G. Andrews, *Concurrent Programming: Principles and Practice*. 637 pages. Benjamin/Cummings. 1991
4.[Bensiya 2002] J. Bensiya and C. G. Davis. A Hierarchical Model for Object Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering*, Vol.28, N°1, January 2002.
5.[Brito 1995] F. Brito, A. M. Goulao, R. Carapuça and R. Esteves, Toward the Design Quality Evaluation for Object Oriented Software Systems, *5th Int. Conf. On Software Quality*, Austin Texas, Oct. 1995
6.[Bruneton 1999] E. Bruneton, J-F. Pradat-Peyre. Automatic Verification of Concurrent Ada Programs. In International Conference on Reliable Software Technologies, Ada-Europe (Ada-Europe'99), *LNCS vol. 1622*, pp. 146-157, Springer-Verlag 1999.
7.[Buhr 1995] P. Buhr, M. Fortier, M. Coffin. Monitor Classification, *ACM Computing Survey*, 27,1, pp. 63-107. 1995.
8.[Burns 1998] Burns A. and Wellings B., *Concurrency in Ada (Second Edition)* Cambridge University Press, 1998.
9.[Burns 2001] Burns A. and Wellings B., *Real-Time Systems and Programming Languages (Third Edition) Ada 95, Real-Time Java and Real-Time POSIX*. Addison Wesley Longmain, 2001.
10.[Cha 1993] Cha, S., Chung, I. S., and Kwon, Y. R. 1993. Complexity measures for concurrent programs based on information-theoretic metrics. *Inf. Process. Lett*. 46, 1 (Apr. 1993), pp. 43-50.
11.[Chidamber 1994] S. R. Chidamber and C. F.Kemerer. A Metric Suite for Object Oriented Design. *IEEE Transactions on Software Engineerin*g, Vol. 20, N°6, June 1994.
12.[Chung 1996] Chung, C., Shih, T. K., Wang, Y., Lin, W., and Kou, Y. 1996. Task decomposition testing and metrics for concurrent programs. In *Proceedings of the the Seventh international Symposium on Software Reliability Engineering (*ISSRE '96) (October 30 - November 02, 1996). ISSRE. IEEE Computer Society.
13.[Clarke 1999] Edmund Clarke, Orna Grumberg and Doron Peled. *Model Checking*. The MIT Press, 1999
14.[Cottet et al. 2002]: F. Cottet, Delacroix J., Kaiser C. and Mammeri Z., *Scheduling in Real-Time Systems*. Wiley Ed., 261 pages, 2002.
15.[Evangelista 2003] S. Evangelista, C. Kaiser, J.F. Pradat-Peyre, P. Rousseau Verifying Linear Time Temporal Logic properties of concurrent Ada programs with Quasar. In *ACM Annual International Conference on Ada (SIGAda'03)*, pp. 17-24, ISBN:1-58113-476-2. San Diego, USA, dec. 2003.
16.[Evangelista 2003] S. Evangelista, C. Kaiser, J.F. Pradat-Peyre, P. Rousseau. Quasar : a new tool for analyzing concurrent programs. International Conference on Reliable Software Technologies, Ada-Europe'03, *LNCS vol. 2655*, pp. 166-181, Springer 2003.
17.[Evangelista 2005] S. Evangelista, C. Kaiser, C. Pajault, J.F. Pradat-Peyre, P. Rousseau Dynamic tasks verification with Quasar. In International Conference on Reliable Software Technologies, Ada-Europe (Ada-Europe'05), *LNCS vol. 3555*, pp. 91-104, Springer-Verlag 2005.
18.[Evangelista 2006] Sami Evangelista, Claude Kaiser, Jean François Pradat-Peyre, and Pierre Rousseau. Comparing Java, C# and Ada monitors queuing policies : a case study and its Ada refinement. *Ada Letters*, 26(2). pp. 23-37, ACM Press, August 2006. (ISSN:1094-3641)
19.[Evangelista 2006] Sami Evangelista, *Méthodes et Outils de Vérification pour les Réseaux de Petri de Haut Niveau. Application à la Vérification de Programmes Ada Concurrents*. PhD thesis CNAM 2006.
20.[GSS 2007] Geotechnical Software Services 2007 *Java Programming Style Guidelines*. 2007 (http://geosoft.no/development/javastyle.html)
21.[Halstead 1977] M. H. Halstead *Elements of Software Science*, New York, Elsevier North-Holland, 1977
22.[Hoare 1974] C. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*. 17,10. pp. 549-557. October 1974

23.[JavaRanch 2007] JavaRanch Project standards. *Java Programming Style Guide* 2007 http://www.javaranch.com/style.jsp
24.[Kaiser 1998] C. Kaiser and J.F. Pradat-Peyre. Reliable, fair and Efficient Concurrent Software with Dynamic Allocation of Identical Resources. pages 109-125, *MCSEAI'98 Congress*, Tunis, December 1998.
25.[Kaiser 2003] C. Kaiser and J.F. Pradat-Peyre. Chameneos, a Concurrency Game for Java, Ada and Others. 8 pages, *ACS/IEEE Int. Conf. AICCSA'03*, Tunis, July 2003. IEEE CS Press.
26.[Kaiser 2007] C. Kaiser, C. Pajault et J.-F. Pradat-Peyre. Modelling remote concurrency with Ada. Case study of symmetric non-deterministic rendez-vous, 12th International Conference on Reliable Software Technologies, Ada-Europe, 2007, *LNCS 4498*, pp. 192-207, Springer 2007. (ISSN 0302-9743)
27.[Kessels 1977] J. L. Kessels. An alternative to event queues for synchronization in monitors. *Communications of the ACM*, 20(7), 500–503. July 1977.
28.[Lea 1999] D. Lea. *Concurrent Programming in Java*. *Design Principles and Patterns*. (*Second Edition*) 340 pages. Addison Wesley 1999.
29.[McCabe 1976] T. J. McCabe. Complexity Measure, *IEEE Transactions on Software Engineering*, Volume 2, No 4, pp 308-320, December 1976
30.[McCabe 1989] T. J. McCabe and C. Butler. Design Complexity and Testing, *Communications of the ACM*, 32, 12, December 1989
31.[Pajault 2006] C. Pajault et J.-F. Pradat-Peyre. Distributed colored Petri net model-checking with Cyclades . In *Proceedings Parallel and Distributed Methods in verifiCation* (PDMC 2006), 2006
32.[Quasar 2007]. http://quasar.cnam.fr/
33.[Rousseau 2006] Pierre Rousseau, *Découpe de programmes concurrents en vue de leur vérification*. PhD thesis CNAM 2006.
34.[Shatz 1988] Sol M. Shatz. Towards Complexity Metrics for Ada Tasking. *IEEE Trans. On Software Engineering*, Vol 14 (8), 1988
35.[Spinellis 2006] Diomidis Spinellis. *Code Quality: The Open Source Perspective*. Addison Wesley, 2006. ISBN 0-321-16607-8.
36.[Tip 1995] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, V. 3 pp. 121–189, 1995.
37.[Xu 2005] Baowen Xu, Ju Qian, Xiaofang Zhang, ZhongqiangWu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2), pp. 1–36, 2005.
38.[Yu 2005] Yuan Yu, Tom Rodeheffer, Wei Chen. RaceTrack: efficient detection of data race conditions via adaptive tracking. *ACM SIGOPS Operating Systems Review*, Vol. 39, 5, pp.221–234, , December 2005.

8. Analyzed programs

All the programs and their Quasar analysis are available on http://quasar.cnam.fr/files/concurrency_papers.html.

9. Annex: Data collected by Quasar

Program name in context5 peer to peer transaction	# task	#Places/ reduced	#transitions/ reduced	#States stored	#Arcs visited	Ratio N/N-1	Execution(s) compile/search	Space (Mb)
agoraserverada3	4	267/70	249/52	4 182	4 369	1	146/0	10.6
agoraserverada4	5	267/70	249/52	138 980	147 901	33.8	159/5	12.4
agoraserverada5	6	267/70	249/52	5 256 768	5 652 471	38.2	421/368	89.6
agoraserverjava3	4	392/93	372/73	8 292	8 633	1	253/1	10.7
agoraserverjava4	5	392/93	372/73	312 619	330 480	38.3	549/18	15.1
agoraserverjava5	6	392/93	372/73	13 140 700	14 027 263	42.4	351/1142	207
agoraserverjavastrong3	4	338/88	318/68	6 542	6 729	1	121/0	10.6
agoraserverjavastrong4	5	338/88	318/68	215 708	224 629	33.4	136/9	13.6
agoraserverjavastrong5	6	338/88	318/68	8 104 338	8 500 041	37.8	138/389	128
agoraserverjavastrongdc3	4	395/100	375/80	8 326	8 565	1	166/0	10.6
agoraserverjavastrongdc4	5	395/100	375/80	280 641	291 914	34.1	162/10	14.6
agoraserverjavastrongdc5	6	395/100	375/80	9 920 178	10 323 753	35.4	148/686	160
agoraserverposix3	4	414/89	396/69	162 912	212 221	1	97/12	12.7
agoraserverposix4	5	414/89	396/69	27 779 147	37 633 949	177	103/4 757	401
rendezvousservermonitor3	5	329/78	308/57	4 739	4 926	1	139/0	10.6
rendezvousservermonitor4	6	329/78	308/57	152 218	160 839	32.7	223/5	12.6
rendezvousservermonitor5	7	329/78	308/57	5 638 789	6 018 940	37.4	148/182	87.2
rendezvousservernested3	5	238/61	219/42	2 824	3 011	1	98 /0	10.6
rendezvousservernested4	6	238/61	219/42	99 040	107 289	35.6	232/6	11.8
rendezvousservernested5	7	238/61	219/42	3 822 728	4 169 915	38.8	262/177	60.3
cooperativeadatasks3	7	550/126	519/95	23 266	24 070	1	356/1	10.9
cooperativeadatasks4	9	550/126	519/95	2 277 347	2 374 070	98.5	363/177	48.5
dummy3	4	225/51	212/38	108	109	1	99/1	10.5
dummy4	5	225/51	212/38	151	153	1.4	107/1	10.5
dummy5	6	225/51	212/38	194	197	1.8	107/0	10.5
dummy6	7	225/51	212/38	237	241	2.2	97/1	10.5

Table 1. Quasar data collected for pairing with peer to peer transaction

Program name in Context1 no pair interaction	# task	#Places/ Reduced	#Transitions/ Reduced	#States stored	#Arcs visited	Ratio N/N-1	Execution(s) Compile/Search	Space (Mb)
xs_agoraserverada3	4	212/59	196/43	3 280	3 145	1	115/0	10.6
xs_agoraserverada4	5	212/59	196/43	89 015	93 844	29.8	191/3	11.6
xs_agoraserverada5	6	212/59	196/43	2 878 524	3 058 587	32.6	115/76	49.8
xs_agoraserverjava3	4	337/83	319/65	10 836	11 206	1	223/0	10.7
xs_agoraserverjava4	5	337/83	319/65	353 350	370 190	33.0	314/10	15.7
xs_agoraserverjava5	6	337/83	319/65	14 215 386	15 082 942	40.7	344/1462	221.3
xs_agoraserverjavastrong3	4	283/76	265/58	5 320	5 455	1	90/0	10.6
xs_agoraserverjavastrong4	5	283/76	265/58	141 842	146 671	26.9	127/4	12.5
xs_agoraserverjavastrong5	6	283/76	265/58	4 526 814	4 706 877	32.1	120/132	76.7
xs_agoraserverposix3	4	360/78	343/59	212 632	279 715	1	85/15	13.4
xs_agoraserverposix4	5	360/78	343/59	42 680 591	58 542 295	209	86/7 640	630
xs_rendezvousservermonitor3	5	274/67	255/48	4 145	4 280	1	144/0	10.6
xs_rendezvousservermonitor4	6	274/67	255/48	110 701	115 530	27.0	238/4	12.0
xs_rendezvousservermonitor5	7	274/67	255/48	3 537 289	3 717 352	32.2	181/142	55.6
xs_rendezvousservernested3	5	183/50	166/33	2 288	2 423	1	79/0	10.6
xs_rendezvousservernested4	6	183/50	166/33	66 556	71 253	29.4	131/2	11.3
xs_rendezvousservernested5	7	183/50	166/33	2 244 376	2 415 091	33.9	134/92	40.0
xs_cooperativeadatasksb3	7	513/121	484/92	25 774	26 578	1	430/1	11.0
xs_cooperativeadatasksb4	9	513/121	484/92	2 538 908	2 635 651	99.2	564/275	53.1
xx_dummy3	4	168/39	157/28	31	31	1	66/1	10.5
xx_dummy4	5	168/39	157/28	32	32	1	71/1	10.5
xx_dummy5	6	168/39	157/28	33	33	1	98/1	10.5
xx_dummy6	7	168/39	157/28	34	34	1.1	102/0	10.5

Table 2. Quasar data collected for pairing with no other interaction

Program name in different extensions with N = 3	# task	# RV	# PO	#Places/ reduced	#transitions/ reduced	#States Stored With reductions	#Arcs Visited With reductions	#States Stored without reductions	#Arcs Visited without reductions
dummy component			**with**	**a**	**shared**			**procedure**	
xx_dummy3	4	0	0	168/37	157/26	29	29	9 608 380	37 388 544
s1_dummy3	4	0	1	192/44	179/30	65	65	16 062 612	6 9421 204
adding a PO				*24/7*	*22/4*	*36*	*36*		
s2_dummy3	4	1	0	205/47	193/35	102	103	13 160 577	50 916 986
s3_dummy3	4	1	1	229/52	215/37	102	103	19 989 509	77 383 894
adding a PO				*24/5*	*22/2*	*0*	*0*		
dummy3	4	2	0	225/47	212/34	103	104	13 312 421	*51 459 540*
t1_dummy3	4	2	1	249/56	234/40	108	109	20 166 873	78 017 848
adding a PO				*24/9*	*22/6*	*5*	*5*		
agoraserverada			**Ada**	**protected**	**object**			**implementation**	
xs_agoraserverada3	4	0	0	212/58	196/42	3 279	3 144	17 316 336	64 230 366
s1_agoraserverada3	4	0	1	236/65	218/46	3 766	3 901	72 406 616	273 741 102
adding a PO				*24/7*	*22/4*	*487*	*757*		
s2_agoraserverada3	4	1	0	249/68	232/51	4 208	4 369	23 798 499	88 896 730
s3_agoraserverada3	4	1	1	273/73	254/53	4 416	4 577	65 691 467	247 715 698
adding a PO				*24/5*	*22/2*	*208*	*208*		
agoraserverada3	4	2	0	267/72	249/54	4 572	4 759	24 563 347	91 637 360
t1_agoraserverada3	4	2	1	293/77	273/56	4 780	4 967	66 789 871	251 631 984
adding a PO				*26/5*	*24/2*	*208*	*208*		

Table 3. Quasar data variation when adding a protected object

Program name in different extensions with N = 3	# task	# RV	# PO	#Places/ reduced	#transitions/ reduced	#States Stored With reductions	#Arcs visited With reductions	#States Stored without reductions	#Arcs Visited without reductions
dummy component			**with**	**a**	**shared**			**procedure**	
xx_dummy3	4	0	0	168/37	157/26	29	29	9 608 380	37 388 544
s2_dummy3	4	1	0	205/47	193/35	102	103	13 160 577	50 916 986
adding a rendezvous				*37/10*	*36/9*	*73*	*74*		
s1_dummy3	4	0	1	192/44	179/30	65	65	16 062 612	69 421 204
s3_dummy3	4	1	1	229/52	215/37	102	103	19 989 509	77 383 894
adding a rendezvous				*37/8*	*36/7*	*36*	*37*		
s2_dummy3	4	1	0	205/47	193/35	102	103	13 160 577	50 916 986
dummy3	4	2	0	225/47	212/34	103	104	13 312 421	*51 459 540*
adding a rendezvous				*20/0*	*19/-1*	*1*	*1*		
s3_dummy3	4	1	1	229/52	215/37	102	103	19 989 509	77 383 894
t1_dummy3	4	2	1	249/56	234/40	108	109	20 166 873	78 017 848
adding a rendezvous				*20/4*	*19/3*	*6*	*6*		
dummy3	4	2	0	225/47	212/34	103	104	13 312 421	*51 459 540*
t2_dummy3	4	3	0	245/55	231/41	114	115	13 464 265	52 002 094
adding a rendezvous				*20/8*	*19/7*	*11*	*11*		
agoraserverada			**Ada**	**protected**	**object**			**implementation**	
xs_agoraserverada3	4	0	0	212/58	196/42	3 279	3 144	17 316 336	64 230 366
s2_agoraserverada3	4	1	0	249/68	232/51	4 208	4 369	23 798 499	88 896 730
adding a rendezvous				*37/910*	*36/9*	*929*	*1 225*		
s1_agoraserverada3	4	0	1	236/65	218/46	3 766	3 901	72 406 616	273 741 102
s3_agoraserverada3	4	1	1	273/73	254/53	4 416	4 577	65 691 467	247 715 698
adding a rendezvous				*37/8*	*36/7*	*650*	*676*		
s2_agoraserverada3	4	1	0	249/68	232/51	4 208	4 369	23 798 499	88 896 730
agoraserverada3	4	2	0	267/72	249/54	4 572	4 759	24 563 347	91 637 360
adding a rendezvous				*18/4*	*17/3*	*364*	*390*		
s3_agoraserverada3	4	1	1	273/73	254/53	4 416	4 577	65 691 467	247 715 698
t1_agoraserverada3	4	2	1	293/77	273/56	4 780	4 967	66 789 871	251 631 984
adding a rendezvous				*20/4*	*19/3*	*364*	*390*		
agoraserverada3	4	2	0	267/72	249/54	4 572	4 759	24 563 347	91 637 360
t2_agoraserverada3	4	3	0	287/76	268/57	4 936	5 149	25 355 511	94 456 846
adding a rendezvous				*20/4*	*19/3*	*364*	*390*		

Table 4. Quasar data variation when adding a rendezvous

Different context names	**#Places/**	**ratio**	**#transitions/**	**ratio**	**#States stored**	**ratio**	**#Arcs visited**	**ratio**
dummy component			**with**	**a**	**shared**		**procedure**	
xx_dummy3 (0 RV, 0 PO)								
no reduction	168	1	157	1	9 608 380	1	37 388 544	1
static reduction only	37	4.5	26	6.0	8 548	1124	28 758	1 300
dynamic reduction only	168	1	157	1	303	31 711	303	123 395
static + dynamic reduction	37	4.5	26	6.0	29	331 323	29	1 289 260
s1_dummy3 (0 RV, 1 PO)								
no reduction	192	1	179	1	16 062 612	1	62 491 204	1
static reduction only	44	4.4	30	6	14 185	1 132	48 147	1 298
dynamic reduction only	192	1	179	1	526	30 537	528	118 355
static + dynamic reduction	44	4.4	30	6	65	247 117	65	961 403
s2_dummy3 (1 RV, 0 PO)								
no reduction	205	1	193	1	13 160 577	1	50 916 986	1
static reduction only	47	4.3	35	5.5	19 081	690	63 071	807
dynamic reduction only	205	1	193	1	632	20 824	633	80 437
static + dynamic reduction	47	4.3	35	5.5	102	129 025	103	494 340
s3_dummy3 (1 RV, 1 PO)								
no reduction	229	1	215	1	19 989 509	1	77 383 894	1
static reduction only	52	4.4	37	5.8	21 861	914	72 483	1 067
dynamic reduction only	229	1	215	1	1 422	14 057	1 430	54 115
static + dynamic reduction	52	4.4	37	5.8	102	195 976	103	751 300
dummy3 (2 RV, 0 PO)								
no reduction	225	1	212	1	13 312 421	1	51 459 540	1
static reduction only	47	4.8	34	6.2	13 053	1 020	42 511	1 210
dynamic reduction only	225	1	212	1	670	19 869	671	76 690
static + dynamic reduction	47	4.8	34	6.2	103	129 247	104	494 803
t1_dummy3 (2 RV, 1 PO)								
no reduction	249	1	234	1	20 166 873	1	78 017 848	1
static reduction only	56	4.4	40	5.8	22 185	909	72 281	1 079
dynamic reduction only	249	1	234	1	1 662	12 134	1 672	46 661
static + dynamic reduction	56	4.4	40	5.8	108	186 730	109	715 760
t2_dummy3 (3 RV,0 PO)								
no reduction	245	1	231	1	13 464 265	1	52 002 094	1
static reduction only	55	4.5	41	5.6	19 705	683	64 607	805
dynamic reduction only	245	1	231	1	708	19 017	709	73 346
static + dynamic reduction	55	4.5	41	5.6	114	118 108	115	452 192
agoraserverada		**Ada**	**protected**		**object**		**implementation**	
xs_agoraserverada3 (0 RV, 0 PO)								
no reduction	212	1	196	1	17 316 336	1	64 230 366	1
static reduction only	58	3.7	42	4.7	91 878	188	268 874	239
dynamic reduction only	212	1	196	1	13 534	1 279	13 669	4 699
static + dynamic reduction	58	3.7	42	4.7	3 279	5 281	3 414	18 814
s1_agoraserverada3 (0 RV, 1 PO)								
no reduction	236	1	218	1	72 406 616	1	273 741 102	1
static reduction only	65	3.6	46	4.7	182 484	397	561 181	488
dynamic reduction only	236	1	218	1	57 282	1 264	58 351	4 691
static + dynamic reduction	65	3.6	46	4.7	3 766	19 226	3 901	70 172
s2_agoraserverada3 (1 RV, 0 PO)								
no reduction	249	1	232	1	23 798 499	1	88 896 730	1
static reduction only	68	3.7	51	4.5	75 223	316	218 483	407
dynamic reduction only	249	1	232	1	16 342	1 456	16 503	5 387
static + dynamic reduction	68	3.7	51	4.5	4 208	5 656	4 369	20 347
s3_agoraserverada3 (1 RV, 1 PO)								
no reduction	273	1	254	1	65 691 467	1	247 715 698	1
static reduction only	73	3.7	53	4.8	123 451	532	369 703	670
dynamic reduction only	273	1	254	1	53 720	1 223	54 567	4 540
static + dynamic reduction	73	3.7	53	4.8	4 416	14 876	4 577	54 122
agoraserverada3 (2 RV, 0 PO)								
no reduction	267	1	249	1	24 563 347	1	91 637 360	1
static reduction only	72	3.7	54	4.6	77 077	319	222 947	411
dynamic reduction only	267	1	249	1	17 980	1 366	18 167	5 044
static + dynamic reduction	72	3.7	54	4.6	4 572	5 373	4 759	19 256
t1_agoraserverada3 (2 RV, 1 PO)								
no reduction	293	1	273	1	66 789 871	1	251 631 984	1
static reduction only	77	3.8	56	4.9	125 593	532	374 887	671
dynamic reduction only	293	1	273	1	56 494	1 182	57 391	4 385
static + dynamic reduction	77	3.8	56	4.9	4 780	13 973	4 967	50 661
t2_agoraserverada3 (3 RV,0 PO)								
no reduction	287	1	268	1	25 355 511	1	94 456 846	1
static reduction only	76	3.8	57	4.7	78 931	321	227 411	415
dynamic reduction only	287	1	268	1	19 722	1 286	19 935	4 738
static + dynamic reduction	76	3.8	57	4.7	4 936	5 137	5 149	18 345

Table 5. Quasar data with different kind of reductions