EXTENDING SOFTWARE COMPLEXITY METRICS TO CONCURRENT PROGRAMS

Flavio De Paoli

Sandro Morasca

Politecnico di Milano - Dipartimento di Elettronica
Piazza L. da Vinci 32 - 20133 MILANO (ITALY)

Abstract
The definition and evaluation of qualities for concurrent software is becoming an important issue, due to the increasing success of distributed and parallel systems. Unfortunately, even in the sequential case, no method is currently available to give a reliable complete assessment of software. Actually, it is unlikely that a unified wide-spectrum theory, covering all different software qualities, will ever be built. Rather, in order to be effective, techniques should address specific characteristics. In this paper, a metric for concurrent software is proposed based on an abstract model (Petri nets), as an extension of McCabe's cyclomatic number. As such, its focus is on the complexity of control flow. Such metric is applied to the assessment of Ada™ programs and an automatic method for its direct computation based on the inspection of Ada™ code is provided.

1. Introduction
In recent years concurrent software systems became increasingly interesting both from a theoretical and from a practical point of view. This is mainly due to advances in hardware, which open new possibilities to computer systems, new programming techniques, and the new application fields for computers. More and more complex computer systems have been built which nowadays play an essential role in many areas where a highly reliable service is needed.

It is rather discouraging to notice that software is the least reliable component in a computer system. This is true in the case of sequential software, but the situation is even worse in the case of concurrent systems. Thus, due to the crucial importance of software in such systems there is need for improving the quality level of concurrent software. But, in the first place, how can we define the concept of quality in an objective way that makes its reasonable to be measured?

Thus, there is a need for strong theoretical bases of software metrics, in such a way that the soundness of the chosen approach and the existence of some properties can be guaranteed. The second need is the case of application of a metric. This issue is in contrast with the first one, in that any significant indicator can come only from a thorough analysis of a program, and in general of software nature.

Such an analysis, as previous experience has shown, is rather costly even for indicators that are not able to reliably grasp software qualities. Therefore, the research should first aim at defining metrics with a satisfactory theoretical underlying support and then at finding, by means of theories and/or approximation techniques - the practical methods for effective measurements. Software is intrinsically complex and thus is highly unlikely that a comprehensive, deep and easy to use theory will ever be built. What we are looking for is the definition of a set of indicators, each of which can cope with a different quality of software, instead of trying to explain all software features within a unique framework. According to this approach, software can be assessed from a set of different viewpoints, so that each indicator brings an amount of information as orthogonal as possible with respect to the others.

A number of theories have been developed in order to devise metrics for sequential software, but the state of the art in the field is still largely unsatisfactory. This is due to some weakness in their theoretical foundations and to the lack of absolute experimental evidence that metrics are actually capable to measuring the features they aim at explaining. As to the concurrent case, only few indicators have been proposed. In [SHATZ 88] an indicator for the communication complexity of Ada™ programs has been proposed. No experiments have been published, owing to the novelty of the proposal.

In this paper we define an indicator to measure the flow complexity of concurrent software. According to what we said, such indicator does not represent a comprehensive approach to a general theory of concurrent software quality. Rather, it is an attempt to provide a precise insight on the problem of defining and measuring a precisely defined software quality, by means of a sound theoretical base. Briefly, it retains characteristics from existing models, namely McCabe's [MCCABE 76] and Shat [SHATZ 88] but it aims at identifying and integrating sequential and concurrent complexities.

The paper is organized as follows. Section 2 is a brief survey of the methods used to measure sequential and concurrent software qualities. In Section 3, after showing how one may represent sequential and concurrent programming languages constructs by Petri nets, we define a complexity metric for Petri nets. The representation of Ada™ by Petri nets is sketched in Section 4, along with a practical method for computing complexity and some examples. The conclusions and an outline of future work are given in Section 5.

2. Metrics for Sequential and Concurrent Programs
Several indicators have been defined to precisely characterize software qualities. Such indicators can be classified into different classes, according to their theoretical background. In particular, indicators measuring a set of software characteristics have been defined based on the number of lines of program code [CONTE et al. 86], on operators and operands present in a program [HALSTEAD 77], and on control and data flow [MCCABE 76] [TAI 84]. The first two approaches can provide useful insights, but the indicators are not based on particularly sound theoretical foundations, and the experimental results do not help in supporting them. On the other hand, control flow based metrics are defined via a well-grounded mathematical derivation and experimental results seem to confirm their usefulness. Unlike Halstead's Software Science, such metrics do not aim at defining a set of indicators that can deal with a large body of software characteristics, but focus only on control flow complexity. However, we have chosen to extend the control flow approach to concurrent software, since we believe that a well-founded approach is definitely indispensable. Thus, in this section we will briefly review the major proposals in the area.

2.1. Control Flow Dependent Metrics and McCabe's Cyclomatic Number
McCabe's approach is rooted in graph theory, and defines software complexity [MCCABE 76] in terms of the so-called cyclomatic number. Such indicator is not defined as a function depending on quantities directly measurable in a program, but is derived from the control flow graph. Thus, it is rather independent of any particular language, and therefore it can be used without changes for programs written in different programming languages. In particular, it abstracts away from the data and focuses only on the control flow of a sequential program.

In graph theory the cyclomatic number v(G) of a strongly
connected graph $G$ (a directed graph where there exists a path connecting every pair of nodes) is the number of its linearly independent paths, i.e., those paths whose linear combinations give all the paths in the graph. If $n$ is the number of edges of $G$ and $e$ is the number of edges of $G$, it can be shown that the cyclomatic number equals $e - n + 1$.

Every sequential program can be associated with a control flow graph, where each node represents a piece of code to be executed in sequence. Nodes representing sequences terminating with a decision point, i.e., a test, have more than two outgoing edges. In a sequential program the start and the end of the execution can be easily identified: in the control flow graph they are respectively represented by a node with no incoming arcs and by a node with no outgoing arcs. Thus, the graph is not strongly connected (the input node is not reachable from any other node): in order to obtain a strongly connected graph an additional edge is introduced from the exit to the start node. Under the additional hypothesis that non-sequational path exists, i.e., a path that is not followed in any execution of the program, the graph is actually strongly connected.

Furthermore, subroutines and functions are to be represented as separated graphs and an edge is added to each of them from the exit node to the input node. The whole graph is therefore the union of $p$ strongly connected components, where $p$ is the number of subroutines and functions including the main program. The control flow complexity of the program is claimed to be the cyclomatic number computed on this augmented graph. Therefore, given the original control flow graph (without the arcs from exit nodes to the input nodes of each connected component) the cyclomatic complexity is computed as follows

$$w(G) = e - n + 2p$$

The construction of the control flow graph could be rather tedious and cumbersome. In order to avoid this difficulty, Mills' theorem proves that the cyclomatic number of each connected component can be obtained also as the number of decision points in the program plus 1 (if the decision point is an n-way decision point it must be counted as $n - 1$ decision points). This synthetic result allows the implementation of simple automated tools for the determination of the cyclomatic number.

2.2. Shatz’s Model

In [SHATZ 88] a metric for the evaluation of Ada is proposed. First it is argued that the complexity of a concurrent program is the weighted sum of two components: sequential complexity, computed as if each task were in isolation, and communication complexity, depending on the interactions among the tasks. Then, communication complexity is defined on the basis of the number of concurrently active rendezvous at any given point of the execution. This model does not take data into account and therefore it can be considered as a control flow approach.

As to the foundations of the model, it is not clear whether it is fully meaningful to separate the two kinds of complexity and then to combine them. This is an open point, as it could be thought that the introduction of concurrency is totally different and unmergeable with respect to the sequential behaviour. Notice that no way to actually compute the coefficients of the weighted sum is provided, and as they should weigh the different importance between the sequential and the communication parts of a concurrent program, one of them can be given a conventional value. Since in the particular case that the program is a sequential program the contribute of the concurrent part of the metric is null, a natural choice for the coefficient of the sequential part is 1.

A major weakness of such model is that it relies too much on a particular, though remarkable, language, i.e., Ada, and does not provide a general way to measure the complexity deriving from concurrency and synchronization in any language.

On the contrary, what really looks interesting is the underlying modelling tool that is assumed in order to represent Ada: Petri nets.

3. A Complexity Metric for Concurrent Software

The brief review outlined in Section 2 has shown how different approaches have been proposed to the problem of assessing software quality. In this section we will show how a metric for concurrent software can be built based on some of them and namely McCabe’s cyclomatic number and Shatz’s model. The choice of such an approach is due to their soundness and ease of use. However, we acknowledge that the limited amount of information that such theories retain from a program restricts their scope of applicability, i.e., they can be used only for complexity measurements.

In the remainder of this Section we will present at first the conceptual model our proposal is based upon, i.e., Petri nets, and the way it can be used to represent a concurrent program. Then, we define a complexity metric for Petri nets which naturally becomes a complexity metric for concurrent programs.

3.1. Basic Concepts of Petri Nets

It is not the aim of this section to provide a fully formal and exhaustive description of Petri nets, but just to recall the most important concepts that will be used throughout the paper. For a wider introduction to Petri nets see [REISER 85] and [PETERTON 81].

A net is a bipartite graph, where nodes are classified in two disjoint classes: places (pictorially represented by circles), which are the passive components of the net and denote transformations of states; transitions (represented by boxes), which are the active components of the net and denote transformations of states. The arcs of the net can connect places to transitions and transitions to places, but not places to places or transitions to transitions. Formally, a net is a triple $(P, T, F)$ where $P$ is called the set of places, $T$ is called the set of transitions and $F$ is called the flow relation, whose elements are the arcs.

The set of places which have an outgoing directed arc into (resp. a transition T is said to be the preset (resp. postset) of $T$. Dual definitions, readily obtainable by exchanging the roles of places and transitions, hold for preset.

A Place/Transition Petri net (P/T net) is a 4-tuple $(P, T, F, m_0)$ where $(P, T, F)$ is a net and $m_0$: $P \rightarrow N$ is an association of a natural number to each place, called initial marking of the P/T net. $m_0$ describes the initial situation of the P/T net. The net can evolve from the initial marking $m_0$ to other markings $m$ through the firing of enabled transitions. In a P/T net a transition $T$ is enabled (with our definition arc weight is always 1) in marking $m$ when the marking of each of the places belonging to the preset of $T$ is at least one. A transition may fire if and only if it is enabled. The effect of the firing of a transition $T$ in a marking $m$ is a change in the marking yielding the new marking $m'$: the marking of each of the places of the preset of $T$ is reduced by 1 and the marking of each of the places of the postset of $T$ is increased by 1.

Starting from the initial marking $m_0$, the net can evolve through different markings produced by the firings of enabled transitions. The marking of a place is represented by putting in the place a number of black dots, called tokens, equal to the value of the marking of the place. Thus, if $m_0(P) = k$, it is usual to say that place $P$ contains $k$ tokens.

3.2. Petri Nets and Basic Constructs of Concurrent Languages

Petri nets are well-suited to represent the features of concurrent software, as well as its sequential part. A token can be used to represent a process: the presence of a token in a place will denote that the corresponding process has reached the state represented by the place it is contained in. The firing of a transition models the execution of a statement or of a sequence of statements. So doing all of the behaviour of concurrent software systems can be captured by means of Petri nets. Here the basic situations that occur in concurrent systems will be graphically represented.

3.2.1. The Sequential Constructs

First of all a sequence is represented in a Petri net as in Figure 1. Each of the three tokens of the initial marking shown in Figure 1 in order to move from place $P_1$ to place $P_3$, has at first to make transition $T_1$ fire.

\[ \text{Figure 1} \]

\[ \text{Figure 2} \]
so that it reaches place P2 and enables transition T2, and then make transition T2 fire so to reach place P3. This is the only path allowed between these two places.

A decision point is represented in a Petri net as in Figure 2, where both transitions T1 and transition T2 are enabled in the marking shown. The token in Figure 2 can choose which transition will fire. In Petri nets, choices are nondeterministic. Thus, a deterministic decision point, as for instance an if then else statement, cannot be exactly represented. This happens also with control flow graphs, since only the skeleton of the program is modeled. The situation in which the firing of a transition disables another transition is called a conflict.

Likewise, it is straightforward to represent a while loop as a Petri net as in Figure 3. Repeat ... until loops and other kind of loops may be represented accordingly.

3.2.2. The Concurrent Constructs

The synchronization between t tasks is represented in a Petri net as in Figure 4. In a "pure" synchronization, the number of arcs entering transition T equals the number of arcs exiting it, i.e. \( t = t' \). In other situations there can be creation or destruction of tasks. In such cases the cardinalities of the incoming and outgoing arcs of T are such that \( 0 \leq t \leq t' \) for creation and \( t \geq t' \) for destruction.

3.3. A Complexity Metric for Petri Nets

A complexity metric can be defined for Petri nets as an extension of the cyclomatic number of ordinary graphs. Before giving its definition we first describe the motivations underlying this proposal.

A sequential program, in that it is a particular case of a concurrent program, can be easily represented by means of Petri nets. In such a case there exists only one place with no incoming arcs, representing the start of computation, and one place with no outgoing arcs, representing the end of computation. The other places have at least one incoming arc and one outgoing arc. As to transitions, every transition in the net has exactly one incoming arc and one outgoing arc. Thus the following relation holds for the Petri net models of sequential systems

\[
|F| = 2|T|
\]

Since in a general Petri net representing a program a transition can have more than one incoming arc and more than one outgoing arc, a measure of the divergence of the behaviour of the system from being sequential can be expressed by the value of the difference

\[
\text{Concurrent Complexity (PN)} = |F| - 2|T|
\]

which can be shown to be nonnegative.

On the other hand, it is true that the cyclomatic number of the ordinary graph representing a sequential program which is isomorphic to the Petri net, can be computed also as

\[
v(G) = |T| - |P| + 2
\]

(we are dealing for simplicity's sake only with graphs with a single connected component) since there is a bijection between the places of the Petri net and the nodes of the graph, and there is a bijection between the transitions of the Petri net and the edges of the graph. Such a value cannot be considered anymore a meaningful indicator for sequential complexity when dealing with concurrent systems, since it is easy to show that it can take negative values. However, due to the equality \( |F| = 2|T| \) in the sequential case, it can be rewritten also as

\[
v(PN) = |F| - |T| - |T| + 2K
\]

We will assume this equation as the definition for the overall complexity of a system represented by a Petri net, thus encapsulating both sequential and concurrent complexity of a program.

The metric can be easily extended in the case of a net composed of K connected components as

\[
v(PN) = |F| - |T| - |T| + 2K
\]

3.4. Remarks

The new metric vs. the cyclomatic number. Like for the cyclomatic number, the complexity measure defined above is fully meaningful if the Petri net is a faithful image of the program: in particular we suppose that no phantom path exists, i.e. every transition in the net is potentially fireable (in Petri net jargon: every transition is 1-live). It is apparent that by construction this measure of complexity is naturally tracked back to the cyclomatic number of the sequential case. Thus, a sequence does not add complexity to the complexity of the overall system and every n-way decision point adds n-1 to it.

As to synchronizations, note that a synchronization point per-se does not add any complexity. In fact, the complexity of a net like the one in figure 4 is equal to the complexity of a net like the one in figure 1. This is intuitively correct: in order for a transition to be fired it is required that the involved tokens (i.e. processes) are together available in the preset of the transition representing the synchronization point, and after the firing they are together available in the postset of that transition. In other words, from the complexity point of view a synchronization point behaves like a sequence. However, this does not mean that synchronizations, when considered within a Petri net, do not add complexity. In the complete representation of a synchronization (cfr. figure 4), the transition is surrounded by the places in its preset and in its postset. On the other hand, when introducing a synchronization point these places already exist, so that only new arcs are added besides the transition.

Bounded and unbounded nets. The new metric has been derived based on topological considerations only, regardless of the marking of the net. This means that we are concerned with the modelling of process types rather than process instances. For instance, if p processes share the code (i.e. they refer to the same process type), their representation is given by a single Petri net marked by p tokens. Anyway, the complexity of the overall system does not depend on the number of processes, but on the different structures of processes involved. The complexity v(PN) we compute on such representation is therefore the intrinsic complexity of the system. A similar approach is used by McCabe when dealing with subprograms. In fact, each subprogram is counted just once, thus considering only its structure.

4. An Application to a Concurrent Language: Ada™

For a general way of modeling Ada™ tasking by means of Petri nets the reader can refer to [MANDRIO, et al. 85]. Here we will adopt a slightly different representation style and will be concerned only with the representation of the major concurrent features of Ada™: accept and entry call statements. The other Ada™ statements related to concurrency can be easily tracked back to similar sequential statements. For instance, the select statement can be viewed as a (possibly nondeterministic) decision point, thus being represented by means of a Petri net conflict.

Ada™ allows the definition of single tasks, task types, and instances of such task types. Since we are interested in their structure, we only deal with single tasks and task types. Instances of task types are represented by means of a suitable marking of the Petri net representation of their task type, but the marking has no effect on our complexity. In what follows, we will use the term task to denote task structure, i.e. single tasks and task types.

4.1. Accept Statement

Figure 5 depicts the Petri net representation of the accept statement. The subnet drawn in black represents the part belonging to the task considered in isolation; the subnet drawn in grey models the interactions with the other tasks. The subnet in isolation is composed of a transition accept_X representing the initial synchronization point,
followed by the net representation of the body, and ending with a transition representing the final synchronization point end接受-X. If the body does not exist, such subnet collapses in just one transition.

In order to deal with the interactions with the other tasks, a place queue-X is introduced which is in the preset of each transition accept-X and a place end_rendezvous-X is introduced which is in the preset of each transition end_accept-X. It must be observed that a Petri net place does not capture exactly the semantics of a FIFO queue. This means that we model executions that are not permitted by Ada™. However, unfeasible executions are present also in McCabe's model, since nothing guarantees that all paths in the control graph are feasible. Moreover, the correctness of concurrent activities must not depend on the arrival order of tasks in the queue: thus, the FIFO policy must have no impact on complexity.

4.2. Entry Call Statement

Figure 6 depicts the Petri net representation of the entry call statement. Again, the subnet drawn in black represents the part belonging to the task considered in isolation; the subnet drawn in grey models the interactions with the other tasks. The subnet in isolation is a sequence transition-place-transition: transition call-t.X models the issuing of the entry call, the intermediate place wait_rendezvous-X the waiting for the completion of the rendezvous, and transition end_wait-X represents the synchronization at the end of the rendezvous. In order to deal with the interactions with the other tasks, an arc is introduced to connect transition call_t.X with the place queue-X belonging to the net representation of task t and end_rendezvous-X with transition end_wait-X belonging to the net representation of task t.

4.3. Direct Computation of Complexity

As in the sequential case, the practical computation of the complexity of a program can be made in a much easier way if the construction of the underlying Petri net which represents an Ada™ program is not necessary. In fact, especially for large size programs, building the corresponding Petri net is a rather tedious and error-prone activity, especially if automated tools are not available. Actually, it is much more effective to compute the complexity of a program only by inspecting its code. This is the aim of Mills’ theorem for McCabe's cyclomatic number.

In this section we provide similar results for computing the overall complexity of an Ada™ program without going through the construction of a Petri net. To this end we will separately examine the effects of the accept and entry call statements on the Petri net representing the program. We will suppose that we already have the representation of each task, in isolation, by means of Petri nets and thus for each task we have already computed the corresponding sequential McCabe's cyclomatic number. Therefore, we will only show the incremental effect of the concurrent statements.

Accept statement. If A(t.X) denotes the number of occurrences of accept X in task t, the effect of the presence of entry X in task t together with its occurrences is given by the difference between the number of new arcs (two for each of its occurrences) and the number of the newly introduced places (queue-X and end_rendezvous-X).

The variation to the overall complexity is

\[ 2(A(t.X) - 1) \]

Entry Call statement. The effect of the entry call statement is twofold. On the one hand new arcs are introduced from transition call_t.X to place queue-X and from place end_rendezvous-X to transition end_wait-X. Thus, defined C(u.t.X) as the number of occurrences of t.X in task u, the effect of the occurrences of t.X due to these arcs is given by

\[ 2C(u.t.X) \]

On the other hand, these new connections reduce the number of existing connected components. The net effect of connecting two previously disjoint components is to decrease by 2 the overall complexity, since the sequential complexity v(t) of each task t is calculated as if it were in isolation. Let us first introduce some notations:

- **Tasks**
- **Entries(t)**: set of the entries of task t
- **Callers(t.X)**: set of tasks issuing entry t.X

Here, we will identify a connected component of the program by means of the tasks it contains: thus, a connected component will be denoted as a subset of **Tasks**.

The algorithm for computing the tasks belonging to the connected components of an Ada program can be sketched as follows.

**ConnectedComponentsSet** := \{ t \mid t \in **Tasks** \};
for each t \in **Tasks** do
for each e in Entries(t) do
for each u in Callers(t.X) do
ConnectedComponent := ConnectedComponent(ConnectedComponentSet);
ConnectedComponent := ConnectedComponent(u, ConnectedComponentSet);
if ConnectedComponent \neq ConnectedComponentSet then ConnectedComponent += ConnectedComponentSet
ConnectedComponentSet \leftarrow ConnectedComponentSet \cup (ConnectedComponentSet \cup ConnectedComponentSet)
fi
od
od
od;

where the function ConnectedComponent(t, ConnectedComponentSet) returns the (unique) element CC of ConnectedComponentSet to which t belongs.

**Definition of the formula.** Let us at first summarize and generalize to the whole program the results obtained for the accept and entry call statements for a single entry and a single task, apart from the effect due to the connections introduced among the tasks. The increment in complexity is given by

\[ \sum_{t \in **Tasks**} \sum_{e \in Entries(t)} (2(A(t.X) - 1) + 2C(u.t.X)) \]

As to the effect of connections, at the beginning when computing the sequential McCabe's cyclomatic number, there are as many connected
components as many tasks belong to the program, i.e. |Tasks|. At the end of the application of the above algorithm, the number of disjoint connected components decreases to |Connected Component|. Thus, the number of connected components that do not exist any longer is computed by the formula:

\[ |\text{Tasks}| - |\text{Connected Component Set}| \]

The complete formula for computing the overall complexity is obtained as the sum of:
- the sequential complexity \( v(t) \) of each task \( t \)
- the increments due to accept and call statements
- the decrement due to new connections.

That is

Complexity = \( \sum\{v(t) - 2\} + \sum\{|2A(t,X) - 1| + \sum\{2C(u,LX)| u \in \text{Callers}(LX)\}| + 2 \) | Connected Component Set |

By means of this formula the overall complexity of an Ada™ program can be easily computed by inspecting its code and recognizing the relevant statements, and counting the number of the connected components.

4.4. Case Study: the Dining Philosophers

As a case study we have chosen the well-known dining philosophers problem proposed by Dijkstra [DIJKSTRA 68]. Five philosophers spend their lives sitting at a round table, thinking and eating spaghetti. To eat spaghetti they need a plate and two forks. Each philosopher has his/her own plate, but there are only five forks: a fork between every two philosophers. Thus, there is competition for fork acquisition: each philosopher must compete with the philosopher at his/her right and with the philosopher at his/her left.

The solution we present here is the one proposed by Gehani in GEHANI 83]. The full code is shown in figure 7. We are interested in the tasks involved in and concurrent activities among them. The main task is the procedure DINING which implements the whole system. Its only activity is to call each philosopher to give him/her an identification number. A task type is defined for philosophers and another one for forks. Each philosopher performs an entry call to acquire his/her right fork and, then, an entry call to acquire his/her left fork. After eating, he/she performs two exit calls to put down the used forks. Each fork task provides two entries to allow philosophers to pick up and put down the fork it controls.

First, we want to show how to compute the complexity on the source code according to the formulas introduced in section 4.3 and then we will compute the complexity of the program by means of Petri nets. Complexity can be computed on the source code by applying the formula:

\[ v(\text{POS}) = \sum\{v(t) - 2\} + \sum\{|2A(t,X) - 1| + \sum\{2C(u,LX)| u \in \text{Callers}(LX)\}| + 2 \) | Connected Component Set |

The task FORK has only one decision point (the select statement with the terminate alternative that provides the exit from the loop). The Mills' theorem (see section 2.1) states that the complexity is the number of decision points plus 1, thus the task FORK has complexity \( 1 + 1 = 2 \). In the same way we can compute the complexity for the other components. The task PHILOSOPHER has a decision point due to the while loop, so that the resulting complexity is 2. The body of procedure DINING is composed of a loop, and again the complexity is 2. The first term of the above formula, computed for all tasks, turns out to be:

\[ \sum\{v(t) - 2\} = (2 - 2)_{\text{fork}} + (2 - 2)_{\text{terminate}} + (2 - 2)_{\text{fork}} = 0 \]  

\( t \in \text{Tasks} \)

The second term deals with concurrent activities, i.e. entry call and accept statements. For each entry we have to count the number of accept statements and the number of entry calls performed by the tasks of the program. Task type FORK is the entry call Accepted PICK-UP, and provides one accept for it. Only tasks of type PHILOSOPHER have rendezvous with tasks of type FORK. Each PHILOSOPHER calls twice the PICK-UP entry. We should remember we are considering

\[ |\text{Tasks}| - |\text{Connected Component Set}| \]

Figure 7

In much the same way, the contribution of the entries FORK.PUT_DOWN and PHILOSOPHER.GET_ID can be computed. Since the program is composed of tasks of three different types which form one connected component, the overall complexity can be computed as follows:

\[ v(\text{POS}) = (0)_{\text{nation}} + (4)_{\text{fork}} + (4)_{\text{terminate}} + (2)_{\text{fork}} + (2)_{\text{terminate}} = 12 \]

Let us compute the complexity on the Petri net representation. Figure 8 shows the whole graph of the program: the concurrent part is drawn in grey. The complexity of the system can be expressed by the cyclical number, which is computed by the formula:

\[ v(\text{POS}) = |T| - |T'| - |T'| + 2 = 54 - 19 + 25 + 2 = 12 \]

As expected, we have obtained the same results in both computations.

5. Conclusions and Future Work

In this paper a metric for concurrent software has been defined as an extension of McCabe's cyclical number. The metric defined here seems to be rather well-grounded and turns out to be suitable in the
However, a wider experimentation is needed in order to better assess its effectiveness. Such experiments will deal at first with a larger and significant sample of Ada™ programs. Then, since the metric is defined independently of any particular language and it is built on top of a general model of concurrency, we will study how to port it on different languages, defining mechanisms for the direct computation of complexity as we did for Ada™. Finally, since Petri nets can be used in software development activities other than coding, like for instance specification, the new metric can be applied to estimate the complexity of code based on the specification or compute the complexity of the specifications themselves.

Acknowledgments

This research has been supported by MPI 40% and CNR (the Italian National Research Council) within "Progetto finalizzato Sistemi informatici e calcolo parall elo" - project "SPECTER".

The authors wish to thank the anonymous referees for their helpful comments.

References