Extending QUASAR with dynamic tasks computation

C. Pajault

Technical Report CEDRIC No695

CEDRIC - CNAM Paris
292, rue St Martin, 75003 Paris
pajault_c@auditeur.cnam.fr

Abstract. The inclusion of dynamic tasks modelisation in Quasar, a tool for automatic analysis of concurrent programs, extends its applicative usefulness. However this extension leads to large size models whose processing has to face combinatory explosion of modeling states. This article is an extension of the paper "Dynamic tasks verification with QUASAR" accepted for publication in the 10th International Conference on Reliable Software Technologies Ada-Europe 2005. It gives some insight into the impact of the way used to name dynamic task in term of state space size by comparing different solutions on several examples.

1 Introduction

As a program structuring entity as well as a distribution support, multi-threading is becoming significantly important in application programs design. Its use is thus being developed in applications which need a high level of integrity and which have to be verified and validated. Nevertheless, interleaving and indeterminism induced by multi-threading introduce a high degree of combinatory that human brain cannot always master. Moreover, concurrent programs are more sensitive to this phenomenon when objects are dynamically created at run time. Using automatic analysis and validation tools is becoming mandatory to obtain reliable concurrent software.

Two years ago, we proposed a tool named QUASAR [EKPPR03] meant to analyze concurrent Ada programs automatically. This first version of QUASAR is relevant to validate critical applications which take care of safety by using a fixed number of tasks and banishing dynamic allocation. This cautious design strategy avoids the explosion of state number and eases certification as well as validation. For this use, QUASAR provides a lot of language constructions related to concurrency such as static tasks declaration, interaction between tasks by means of rendez-vous, protected objects or shared variables.

As an increasing number of concurrent applications programs use dynamic task allocation and object-oriented-programming (which relies on object dynamic allocation), we decided to augment QUASAR usability by allowing it modeling dynamic task allocation, even if this extension may involve the risk to

generate a large number of application states and to have to face combinatory explosion. However we were confident in the QUASAR approach in so far as it reduces the application state numbers as soon as possible in the verification process.

Compared to other works in the domain [MSS89], [SMBT90], [NM94], [BW99], [BWB+00], [BBS00], our approach is fully automatic. The analysis of concurrent programs is performed in four steps.

First, the original source code of the program is automatically sliced in order to extract the part of the program that is relevant to the property the user wants to check. Safety properties of concurrent programs concern the absence of deadlock or of livelock, the coherence of shared variable, 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.

Second, the sliced program is translated into a colored Petri net 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. For example, the loop pattern contains a meta-net corresponding to the pattern of sequence of statements. 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.

In the third step QUASAR checks the required property on the Petri net, using successively graph reduction and structural techniques or state based enumeration techniques (model-checking).

At last, if the required property is not verified, the user is provided with a report demonstrating a sequence invalidating the property.

A detailed description of this process can be found in [EKPPR03], [EKPPR04] or at the url `quasar.cnam.fr`.

This paper describes the new feature that makes QUASAR suitable for automatic analysis of dynamic concurrent programs and is organized as follows.

Section 2 presents briefly Ada dynamic tasks semantic and their synchronization constraints at elaboration and completion stages.

Section 3 presents the choices that we made to model the dynamic tasks together with the static tasks in QUASAR and describes the Petri nets library patterns corresponding to dynamic allocation and to operations on pointers. We justify our choices regarding the solution by evaluating the consequences on the analysis step.

At last, Section 4 highlights this new feature by analyzing two significant Ada programs that use dynamic tasks, and by displaying the state number of each model. This shows that the state explosion has been limited by our design choices and that automatically validating significant concurrent programs using dynamic allocation is a feasible enterprise.

2 Ada Tasks

In Ada the unit of parallelism, the sequential process, is named a task. Concurrency between tasks may be achieved via task rendez-vous, via protected objects or via global variables (possibly further qualified with volatile and atomic pragmas). As these features belong to the language, and have a precise semantic definition, the tasks behavior control is not dependent on compiler or operating system choices as it is the case with application programmer's interfaces (even if standardized as POSIX API).

In order to analyze concurrent programs, QUASAR models the tasks behavior and their interactions. As we assume that the concurrent programs have been designed following the Ada 95 Reference Manual [Int95] and are free of compiling errors, QUASAR can focus on the regular states reached by Ada tasks during their lifetime.

In this section, we will briefly describe task declaration and task lifetime. A more detailed description can be found in [BW95], [Dil93], [Dil97], and naturally in the Ada Reference Manual [Int95].

2.1 Task Declaration

A task declaration has two parts : the interface (or specification part) and the body.

The body corresponds to the code executed by the task at run-time. The interface is the visible part of a task and is used to specify the *entries* of the task.

A task entry can be remotely called by other tasks to synchronize, to send informations, or to communicate with the task owning the called entry. The calling task and the called task cooperate by rendez-vous.

2.2 Task creation

Tasks can be created in two ways. The first way is static creation : one can declare a task type describing the task and then declare a variable of this type. The other way is dynamic creation : it uses *access types* (pointers).

The program Figure 1 illustrates the two ways of creating tasks :

- line 2 shows a task type declaration ;
- line 20 shows a static instance declaration; such a task is called a **static** or an **elaborated** task ;
- line 10 shows a dynamic instance creation of a given type ; the access type is declared at line 3 and the pointer is declared at line 7 ; such a task is called a **dynamic** or an **allocated** task ; the *new* clause refers to a generic storage allocator used for the creation of the task.

This paper discusses how QUASAR copes with dynamic task instantiation.

```
1 procedure ALLOC is
2    task type TT;
3    type OUTT is access TT;
4
5    procedure P is
6       type INTT is access TT;
7       O : OUTT;
8       I : INTT;
9    begin -- body of P
10      O := new TT; -- allocated / dynamic task
11      I := new TT; -- allocated / dynamic task
12      ...
13      -- procedure P completion
14   end P;
15
16   task body TT is
17      ...
18   end TT;
19
20   T : TT; -- elaborate / static task
21
22 begin -- body of ALLOC
23    P; -- call P
24    ...
25    -- ALLOC completion
26 end ALLOC;
```

Fig. 1. Ada program with task declarations and creations

2.3 Task hierarchy

Ada is a block structured language in which blocks may be nested within blocks. Ada distinguishes between declarations and statements. At compile time, the declaration of an entity declares the entity. At run time, the declarations are elaborated and this elaboration creates the declared entities.

A task can be declared in any block and this creates a task hierarchy. A task which is directly responsible for creating another task is the *parent* of the task and the created task is called the *child*.

The creation and termination of tasks within a hierarchy affect the behavior of other tasks in the hierarchy.

The parent of a child task is responsible for the creation of that task. The *master* of a *dependant* task must wait for that task to terminate before it can itself terminate. The parent of a task is often the master of that task, however with dynamic task creation the master is not the parent task but the task that contains the declaration of the access type.

More precisely [Dil97], every task instance in program state, except the root instance created for the main procedure, has exactly one direct master and possible indirect masters, as described by the three following rules:

[**Master 1.**] The parent of an elaborated task instance is also the direct master of the elaborated task instance.

[**Master 2.**] The instance that declares the access type of an allocator expression is the direct master of all allocated tasks instances that are created by evaluating the expression.

[**Indirect Master.**] An instance is an indirect master of a task instance if either (1) it invokes (directly or indirectly) a master of the task instance, or (2) it is a master of a master of the task instance.

To illustrate this notion, let's consider Figure 1. The first rule implies that the parent of task `T` elaborated at line 20 is also its master. As the parent of this task is the procedure `ALLOC`, it is also its master. The second rule implies that, even if the pointer `O` is allocated by the procedure `P` at line 10, since the access type `OUTT` is declared by the procedure `ALLOC`, this procedure is the master of the task created line 10. The third rule implies that `ALLOC` is an indirect master of the task `I`, since it invokes `P` which is a direct master of `I`.

2.4 Task activation, execution, finalization and termination

Using these rules we can now present the synchronizations induced by these dependences at each stage of a task lifetime.

- *Activation.* This state corresponds to the elaboration of the declarative part of a task, the creation and initialization of the local variables of the task. A task cannot achieve its activation phase as long as all tasks declared in its declarative part have not finished their own activation phase. This is the first synchronization point. A child activation starts at the end of its parent task elaboration and proceeds concurrently with its parent task.
  Note that in the case of a dynamic task, this task is activated as the last step of its creation and the parent of an allocated instance is blocked during the call of the task allocator until the instance finishes activating.
  Note that a task entry can be called before the task has been activated.
- *Execution.* This state corresponds to the execution of the task body. No synchronization is induced by task dependences.
- *Completion.* This state corresponds to the state of a task in which all the statements of its body have been executed. A task cannot leave this state and start its termination step if the tasks depending on it have not yet finished their termination. This is the second and last synchronization point.
  Note that during completion, entries of the task can be called.
- *Termination.* This state corresponds to the destruction process of the task. No synchronization is induced by task dependences. According to the second and last synchronization point defined above, a task instance enters this state when it is completed and all of its dependent tasks are terminated.

We have presented the definitions of Master and of different dependence rules for tasks, but they apply also for procedures, functions and any kind of blocks.

3 Modeling tasks

We consider now how to map an Ada task into a colored Petri net. The generic pattern modeling a component that includes both declarations and statements (e.g. a sub-program, a task, or a block statements) is composed of four "meta-transitions" (i.e. abstract transitions that will be replaced by more concrete ones) and by five intermediate states (places `C.Begin`, `C.Ready`, ..., `C.End`). It is depicted Figure 2.

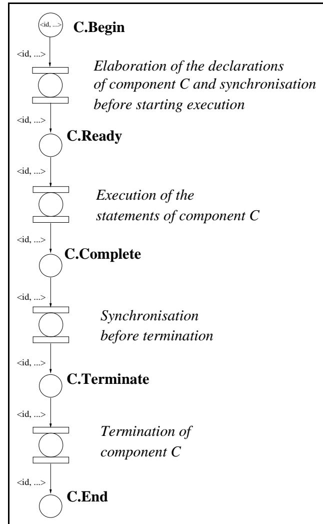

Fig. 2. Component model in QUASAR

In order to analyze concurrent programs including dynamic tasks we have to take into account the dependences existing between a "spawned" task and its master. Several difficulties have to be solved : the dynamic naming of a task, the dynamic referencing of the master, the dependence synchronizations modeling.

3.1 Dynamic *id* computation

In order to model tasks and specially their synchronization, a unique model *id* must be assigned to each task of a program execution being analyzed by QUASAR .

When all tasks are elaborated in the program, the number of tasks is constant and is known at compile time. Thus, an *id* can be assigned statically and it is the same for all possible executions of the program.

When the number of tasks is variable and may vary from one execution to another, task modeling must generate *ids* dynamically and this implies that a task *id* may vary from an execution to another. Having to care of it adds complexity to the model. The challenge is then to find a way of generating *ids* which change as little as possible (and ideally that does not change) from one execution to another.

Three solutions were considered. They are depicted in the next three parts. To illustrate these solutions, we use a small example presented figure 17.

Global *id* server The first solution is to manage a "global *id* server" such that each created task (elaborated or allocated) is given an *id* by the "*id* server". The main drawback of this solution is the combinatory implied by the global server.

Indeed, for different executions of a program, a task can get different *ids*. As all the possible executions are considered at the verification step, this leads to combinatory explosion.

To illustrate this, Figure 3 presents a colored Petri net representing a part of the translation of the program of figure 17. This net is simplified as we only want to focus on the dynamic identification. Tasks elaborated at line 25, 26 and 27 of the program are represented by tokens in the places `T1.BEGIN` and `T2.BEGIN`. The first value in the token is the *id* of the task, the second value represents the values present in each task token. With this simple topography, we can assume that the combinatory generated will be high.

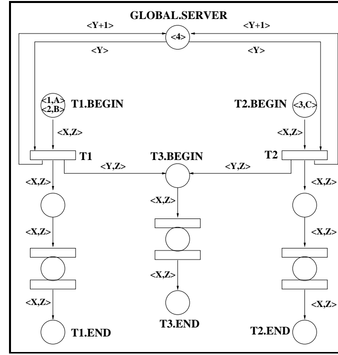

Fig. 3. Part of the net generated for the program of figure 17. We use the global *id* server as identification mecanism.

When modeling the example program, this solution has to cope with six different *ids* for each dynamic task. As recorded Table 1, the corresponding model generates a large number of states in the accessibility graph.

Local *id* server A second possibility is to use "local *id* servers". A local server is associated with each statement in the program that may spawn tasks. As the number of statements that spawn tasks in the program is known, we are able to give each local server a unique and statically determined *id*. Each local server thus has a statically determined unique *id* and assigns a new *id* to tasks spawned by its corresponding statement in the program. Task identification is therefore done with two values : the *id* of the local server and the value this server gives to the task. This solution allows to consider the combinations at a local level and highly reduces combinatory as compared to the global server. This is a solution similar to the known "divide and conquer" strategy for program complexity reduction.

To illustrate this, let's consider the program presented Figure 17. In this program, five statements at lines 15, 22, 25, 26 and 27 are statements or declarations that spawn tasks. So the generated net will contain five local servers. If we focus

on the tasks `T1`, `T2` and `T3`, the generated net will look like the net presented in Figure 4. The first local server is for the `new` statement of line 15, the second one is for the `new` statement of line 22. The first value of the task token is the *id* of the local server that gives the task its *id*, the second value is the *id* given by this local server and the last value represent the different values of the task.

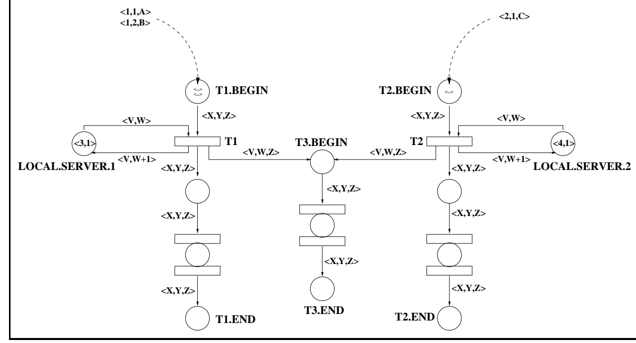

Fig. 4. Part of the net generated for the program of figure 17. We use the local *id* server as identification mecanims.

When modeling the example program, this solution has to cope only with two different *ids* for each dynamic task. As shown Table 1, combinatory is highly reduced comparing to the global server solution.

One-to-one function A third choice is to use a one-to-one mapping. For each task, we keep a value n that is modified at each task creation (using for instance the number of created tasks, or the *id* of the last created task). The *id* of a spawned task is then computed using a hashing function. For example, a unique *id* is computed using the *id* of the father and the current father's value of n. Each assigned *id* in use is kept in a special place. In case of collision we choose the first *id* greater than the one computed (thus we obtain a one-to-one mapping). In the absence of collision, the *id* assigned to a task is not dependent of the interleaving, and thus, no combinatory is induced by the dynamic task internal naming.

We have chosen to implement this last solution since it is the one that minimizes combinatory in practice as we can see in table 1.

To find a good hashing function, that is a function that avoids collisions, we can distinguish different cases:

– if all tasks are spawned by a single block, a one-to-one function is defined by:
$$f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$
$$f(id,n) = id + n + 1$$

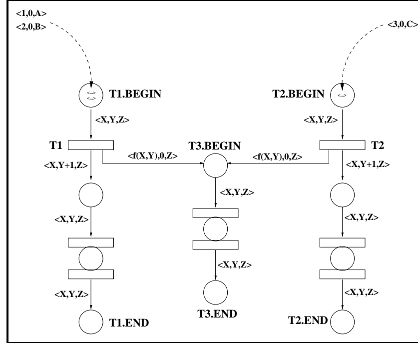

Fig. 5. Part of the net generated for the program of figure 17. We use the one-to-one function as identification mecanims.

id allocation mecanims	*Number of generated states*	*Number of different name sequences*
Global *id* server	493	6
Local *id* server	205	2
One-to-one function	125	1

Table 1. Combinatory evaluation for the three different *id* allocation mecanims for the nets presented on figures 3, 4 and 5.

where *id* is the *id* of the parent task and n is the number of tasks already spawned by the parent task.

- if the number of tasks that the program may spawn can be raised, we are able to find a unique identification function that avoids collisions. This function is defined by:

$$f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$
$$f(id,n) = id + n + max$$

where *id* is the *id* of the parent task (and $id \geq 1$), n is the *id* of the last task created by the parent and *max* is the maximum number of tasks that may be spawned by the program ($max \in \mathbb{N}$).
- if a program does not fit into one of these models, we have to write a specific one-to-one function.

3.2 Referencing the master of a task

For the sake of simplicity we have decided to implement differently allocated and elaborated tasks and to generate one net for the task type and one net for each access type. For instance, the model corresponding to the program depicted Figure 1 will contain three different nets issued from type `TT`: one for the type itself, one for the type `INTT` and one for the type `OUTT`.

In order to retrieve the master of a task we have to store partially the task hierarchy. This hierarchy is specific to every block that declares an access task type. Two different ways of modeling this hierarchy may be used, either using specific places or using a more complex token for the task.

Modeling a part of the task hierarchy with places First, we can store the master of an allocated task in a specific place for each block declaring an access task type. This place will contain tokens $\langle\ id, id_master\ \rangle$ where *id* is the identifier of an allocated or an elaborated task and *id_master* is the identifier of the master of the corresponding block. Then, each time an allocated task ends it can use this place to retrieve its master *id*.

For the program Figure 1 we have to define two places: one for the block `Alloc` (we denote it p_A) and one for the procedure `P` (we denote it p_P). When a task id_0 (may be the main task) calls procedure `Alloc` it puts a token $\langle\ id_0, id_0\ \rangle$ in p_A. The declaration of `T` leads to put a token $\langle\ id_T, id_0\ \rangle$ in the place p_A (where *id_T* is the *id* assigned to `T`); so `T` may allocate tasks of type `OUTT` and give them the correct master. When id_0 calls P, it puts a token $\langle\ id_0, id_0\ \rangle$ in p_P. The allocation of `O` puts the token $\langle\ id_O, id_0\ \rangle$ in both places p_A and p_P since these two blocks declare access task type visible from `O`. The same operation occurs when allocating `I`.

The advantage of this solution is to simplify the task token and this may optimize the firing rule and the memory used to store the state space when the model is analyzed. The drawback is that this solution may complicate synchronization patterns and then may result in a less efficient reduction ratio in the reduction step (occurring before the analysis).

Modeling a part of the task hierarchy in the task token The second solution consists in adding an *id* table in task tokens and in giving a part of the execution history to created tasks so that they can find their master. The table size is fixed at compile time and is set to the number of blocks that declares access types on task types in the whole program. For each one of these blocks which is a possible master of some allocated tasks, a specific entry is added in the table.

In the net, when a block declares an access type, it puts its *id* in the corresponding entry of the table. When a block spawns a new task or calls a new block, it gives it its *id* table. The idea is to propagate the table so that when an allocated task has to find its master, it just has to catch the *id* at the corresponding entry of the table.

To illustrate this, let's consider the example presented on Figure 1. Procedure `ALLOC` declares the access type `OUTT`, so an entry is added to the table. Procedure `P` declares the access type `INTT`, so a second entry is added. As there is no more access type declaration in the program, table size is two: the first entry is for procedure `ALLOC` *id* and the second one for procedure `P` *id*. In the generated net, when procedure `ALLOC` calls procedure `P` at line 27, it gives it its table value (that is, [*alloc_id*,*null*]). Then, when `P` declares its access type at line 6, it adds

its *id* in the corresponding entry. When P spawns O and I, it gives them its table ([*alloc_id*,*p_id*]). As the master of O is procedure ALLOC, O has just to catch the right *id* in the corresponding entry of the table (that is, the first entry) and as the master of I is procedure P, I just has to catch the *id* in the corresponding entry (that is, the second one).

The main drawback of this solution is that all tokens have to keep this table even if they will not use it. Moreover, if many access types are declared in many different blocks, the size of the table will be large and state representation will need more memory. A good way to limit this increase is to group the access type declarations in a single block as much as possible. By this way, the table size will remain small, leading to a better efficiency of Quasar .

3.3 Tasks management patterns with hierarchy in places

An allocated task depends on its master which is not necessarily its father but is always an ascendant. In order to manage this dependence we introduce a place in the generic model of blocks containing an access task type declaration, the place `C.Id_Master`. This place is meant to contain tokens that are pair of identifier $\langle\, id, id_master \,\rangle$ where *id_master* denotes the master of the task *id*. The second place we introduce is the place `Task_Type_Name.Dependences` which notes the number of allocated tasks that the master has to wait for when it's ending. There is one place for each access task type model. This place contains tokens $\langle\, id_master, cpt \,\rangle$ where *id_master* is the process identifier of the task that has elaborated the block containing the declaration of the access type and *cpt* counts the number of tasks of this type which is being active at a given stage.

When a task *id* (may be the principal one) elaborates a component C containing the definition of a task access type (see Figure 12, transition t1) a token $\langle\, id, 0 \,\rangle$ is produced in the place `Task_Type_Name.Dependences` where `Task_Type_Name` is the name of the type. This transition initializes the counter associated with this type by noting that, at this stage, C does not depend of any task. This firing is also used to record that the task *id* is the master of this type by putting a token $\langle\, id, id \,\rangle$ in the place `C_Name.Id_Master`. The task that has elaborated the component C cannot reach the state `C.Terminate` until the number of active task depending on it equals zero (transition t2).

When a task is created (with an allocation or a declaration) it has to be aware of the master of each visible access task type (we know them at the net generation). For doing that we use the following pattern that prepares the allocation or the declaration (see Figure 7). Its role is twofold : first it computes a new identifier for this child (`id_child`, transition t1). Second (transition t2), it notes that this new task can allocate tasks of every visible access task type. So, a token $\langle\, id, id_i \,\rangle$ is put in each place `Ci.Id_Master` where `Ci` denotes a block declaring access task type which is visible from the created task. The master of this component is known by the token $\langle\, id, id_i \,\rangle$ present initially in this place.

Conversely, at task end, all tokens put in places `Ci.Id_Master` during the creation preparation have to be removed. It's the role of the following pattern (see Figure 8).

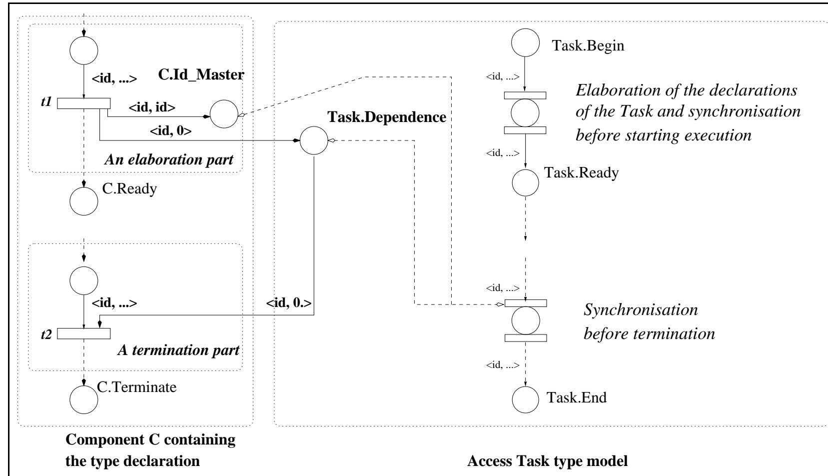

Fig. 6. Elaboration of a component containing an access task type definition

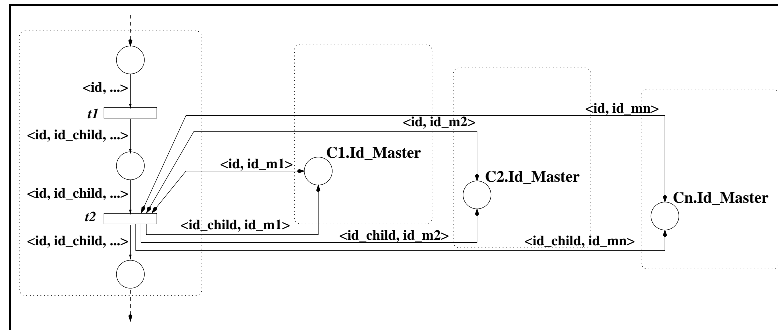

Fig. 7. Preparation of a dynamic task creation

For the allocation of a new task (see Figure13) we first have to prepare the creation (using the pattern described before in figure 7). Then (transition t1), the number of tasks on which the block that elaborated the type of the allocated task (by the execution of the task *id_master*) depends is incremented by modifying the token $\langle\ id_master, cpt\ \rangle$ into the token $\langle\ id_master, cpt+1\ \rangle$. Then (transition t2), the new task is started by putting a token $\langle\ id, id_child\ \rangle$ in the place `T.Begin`. At last (transition t3) the creating task has to wait until the allocated task has finished its elaboration.

The termination of an allocated task (see Figure10) consists in preparing this termination (using a previously depicted pattern) and then in decrementing the counter associated with the access task type.

The creation of a task by the evaluation of a declaration follows the pattern depicted Figure 14. First, the creation is prepared as previously. Second, the created task is activated and the creating task waits until the end of its elaboration.

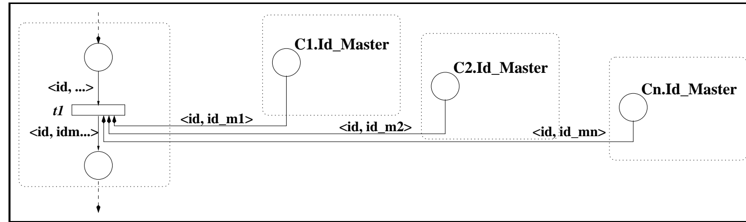

Fig. 8. Preparation of a task termination

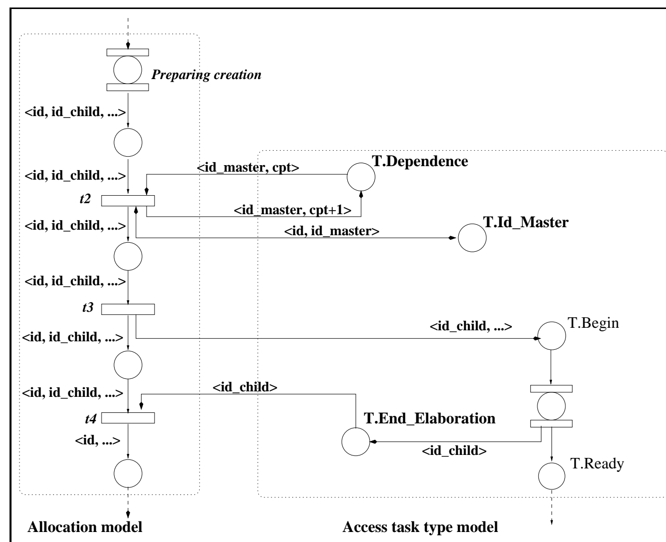

Fig. 9. Evaluation of a task allocator

At the end of the created task, this one signals its end by putting a token in the place `T.End_Execution`. This token allows the task that declared it to continue (transition t2).

3.4 Tasks management patterns with hierarchy in task token

We now define the patterns that we use to generate colored nets for Ada programs that contain dynamic tasks creation. The first pattern concerns the elaboration of a block containing a definition of an access task type and is depicted Figure 12. It acts as follows : when a task id (may be the main one) elaborates a component containing the definition of an access task type, a token $\langle\ id, 0\ \rangle$ is produced in the place `Task_Type_Name.Dependences` where `Task_Type_Name` is the name of the type. This transition initializes the counter associated with this type by noting that, at this state, `C` does not depend of any task. The task that has elaborated the component C cannot reach the state `C.Terminate` until the number of active task depending on it equals zero (transition t2).

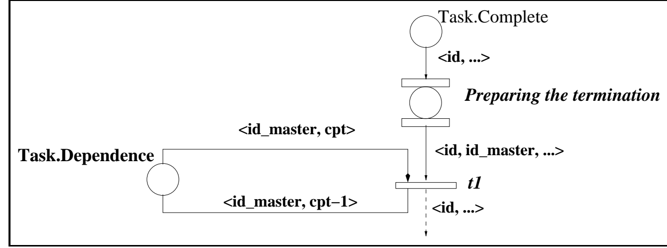

Fig. 10. Termination of an allocated task

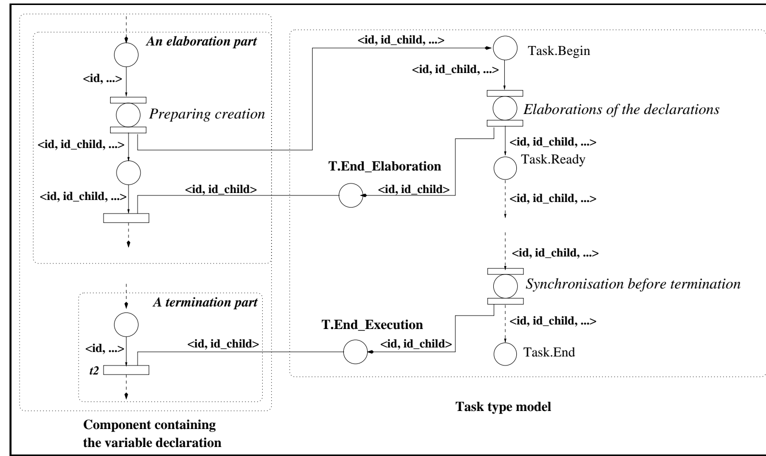

Fig. 11. Creation of a task by the evaluation of a declaration

For the allocation of a new task we use the pattern depicted Figure 13. First we increment the number of tasks on which the block that elaborated the type of the allocated task depends and therefore modify the token $\langle\ id_master, cpt\ \rangle$ into the token $\langle\ id_master, cpt + 1\ \rangle$ in the place `Task.Dependence`.

Then, with transition t2, we start the new task by putting a token $\langle\ id, id_child\ \rangle$ in the place `T.Begin` (the identifier of this new task is computed as described in a previous section). At last, at transition t3, the task waits until the allocated task has finished its elaboration. The place $V.Var$ is used to store that task id allocate task id_child and to synchronize the termination of a task with its father.

The creation of a task by the evaluation of a declaration is done according to the pattern depicted Figure 14. The created task is activated and the father waits until the end of its elaboration. At the end of the created task, this one signals that it has finished by putting a token in the place `T.End_Execution`. This token allows the task that declared it to continue (transition t2).

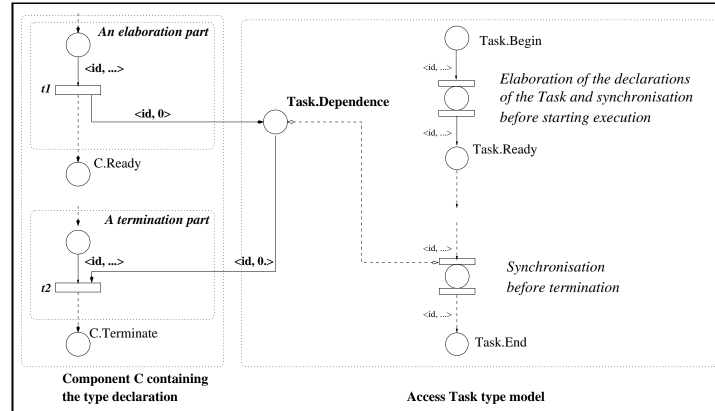

Fig. 12. Elaboration of a component containing an access task type definition

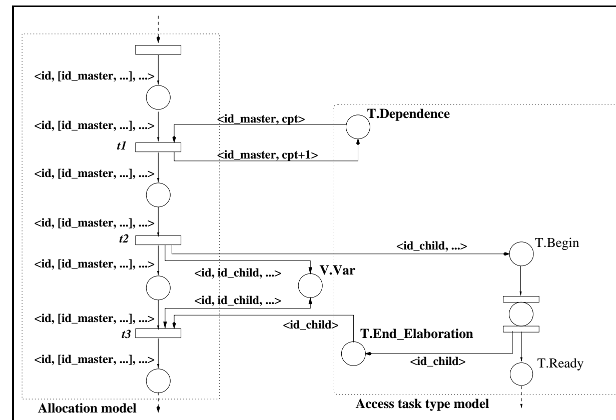

Fig. 13. Evaluation of a task allocator

3.5 Evaluation

We evaluated the two different ways of modeling a part of the task hierarchy. We checked the two ways using the sieve of Eratosthene (Figure 15). Results are presented Table 2.

Results could be partially explained by the termination preparation of the places version. Compared to the token solution, the place version adds another transition in the model, and this generates additional combinatory.

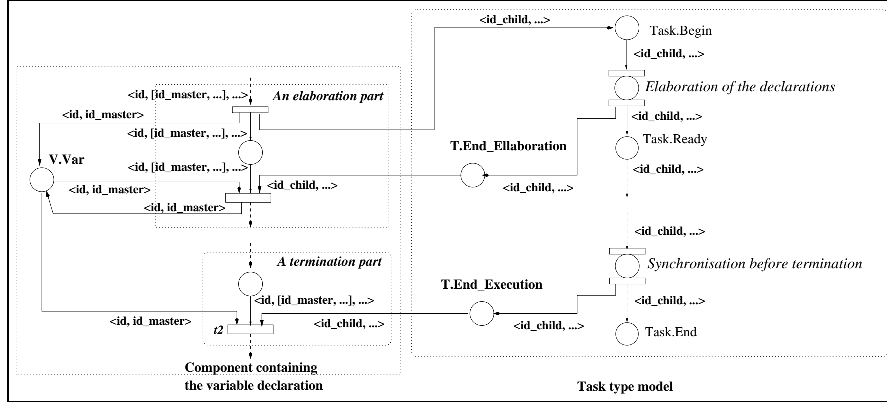

Fig. 14. Creation of a task by the evaluation of a declaration

	Token *version of* Quasar		**Places** *version of* Quasar	
Clients	*States without reductions*	*States with Reductions*	*States without reductions*	*States with Reductions*
1	2 549	247	2 699	246
2	438 913	9 499	536 335	12 357
3	–	735 767	–	1 347 009

Table 2. Evaluation of the two methods for modeling dependences on the client/server program presented in Figure 16.

4 Cases studies

4.1 The sieve of Eratosthene

The sieve of Eratosthene is used to find all primes inferior to a number N. It considers each number between 2 and N and eliminates all the multiples of each considered number. We have implemented a concurrent version of it.

For each prime number, a new task is spawned. The program can be viewed as a dynamic task pipeline, as a task can be added dynamically to the pipe. Each stage of the pipeline is a task with a prime and represents a part of the sieve. The algorithm is presented Figure 15. The main procedure is a loop which calls the entry of the first `Prime` task of the pipeline with each number to sift. At the end of the loop, it sends the termination message to the pipeline. `Prime` task program is a loop which iterates until the termination message is received. The task waits on its entry `Test_Primality`, stores the received number in a buffer and ends the entry call. Then, it checks the primality of the received number; if the task has not stored any prime number yet, it keeps the received number as its prime and waits for another call on its entry. If the task already stores a

prime number, it checks whether it can divide it, if not, it creates another `Prime` task (if it does not already exists) by calling the function `Create_Prime` and send it the number to be tested. In this example, we don't collect the primes since we are interested in the dynamic tasks creation and termination only.

Results of our experimentations are presented Figure 3. Given N, we indicate

N	*Running tasks*	*States without reductions*	*States with reductions*	*Reduction factor*
3	3	1 556	294	5
6	4	19 160	1 784	11
10	5	224 102	10 047	22
12	6	2 810 870	65 645	43

Table 3. Evaluation for the sieve of Eratosthene

the number of running tasks and we calculate the number of states generated by the model both with and without reductions. Evaluation data were obtained with our model-checker Helena [Eva04].

4.2 A Client/Server example

The program presented Figure 16 in the Appendix is a simple client/server program example. The main loop spawns clients of the task server. For each calling client, the task server creates a `Thread` task by allocating an `Access_Thread` pointer and by returning the pointer to the client. The client thus calls the entry of this `Thread` task which accesses the protected object managing the data. For different numbers of clients, we calculate the number of generated states both with and without reductions and also the number of states relevant to the deadlock free property after slicing and reductions. Results are presented on table 4.

Clients	*Running tasks*	*States without reductions*	*States with reductions*	*Slicing & reductions*
1	4	2 549	247	221
2	6	438 913	9 499	5 939
3	8	–	735 767	239 723
4	10	–	–	12 847 017

Table 4. Evaluation for the Client/Server program

Results show that even when the integration of dynamic tasks and synchronization mechanisms implies a great combinatory, QUASAR is able to reduce highly the generated model via slicing mechanisms and structural reductions of the Petri net.

4.3 Further remarks

Let us now modify the previous program so that it contains run time errors and let us show that these errors can be easily detected by QUASAR . In an execution without deadlocks, each task token reaches the place END of its Petri net model. So, if in a terminal state, a task token is in another place, this means that there exists an execution leading to a terminal state which corresponds either to a deadlock or to a run-time error.

In QUASAR , when a task calls an entry, it puts a token with its *id* and the *id* of the "server" task in an entry call place and gets the results from an entry return place. The "server" task takes the token in the call place, gets the client's *id* and notifies it in the return place when it has finished the call. Then the client can continue its execution and reach its place END.

Suppose that the task `Task_Server` loops from 1 to `MAX_CLIENT-1` only, the server task will never take the token of the last client in the call place. So, this client will be blocked and will never reach the place END. The detection of such a deadlock is very easy in QUASAR .

Imagine now a second case of error when the variable `The_Task_Server` is of access type. Suppose that this access variable is allocated by the main procedure. When a client calls the entry of this dynamic task, two cases are possible: in the first one, the client calls the entry of `The_Task_Server` before this dynamic task has been allocated and it is blocked; this leads to a deadlock. In the second case, the client calls the entry after the allocation and the entry call is accepted.

The behavior of such a program depends on the order in which allocations are performed and this kind of error is not detected by the compiler. As QUASAR enumerates each possible execution, it points out this kind of behavior. When an access variable is declared, it gets a special initialization value set to the `null` value. Then, if the client calls the entry before the variable has been allocated, it is blocked in the entry call place because of a wrong *id* for the variable `The_Task_Server`. Its entry call is never accepted and it never reaches its END place. This causes a deadlock which is detected by QUASAR .

5 Conclusion and Future works

This paper shows that dynamic allocation can be modeled by tools that make use of Petri nets. The use of both this modeling and the QUASAR techniques allow to limit the model states explosion, making automatic validation of dynamic concurrent programs feasible. Moreover it has been shown that QUASAR is able to detect synchronization errors which occurrence depends on the dynamic task allocation order and which is not detected at compile time.

The scope of application programs that can be verified by QUASAR can still be extended, and in particular to object oriented programming as it is approached for Java programs in [NAC99], [CDH+00], [BR01], [FLL+02], or for other concurrent languages as in [AQR+04] or in [HRD04].

Integrating these concepts involves a high level of combinatory. We are now considering this problem : on the one hand, we are already working on Helena [Eva04], a new model checker which allows colored Petri nets reductions and on the other hand we will further increase the analysis power by studying parallel and distributed model checking algorithms.

References

[AQR+04] T. Andrews, S. Qadeer, J. Rehof, S.K. Rajamani, and Y. Xie. Zing: Exploiting program structure for model checking concurrent software. In *Proceedings of the 15th International Conference on Concurrency Theory*, 2004.

[BBS00] J. Blieberger, B. Burgstaller, and B. Scholz. Symbolic Data Flow Analysis for Detecting Deadlocks in Ada Tasking Programs. In *Proc. of the Ada-Europe International Conference on Reliable Software Technologies*, Potsdam, Germany, 2000.

[BR01] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free java programs. *SIGPLAN Not.*, 36(11):56–69, 2001.

[BW95] A. Burns and A. Wellings. *Concurrency in Ada*, chapter 6.11, pages 134–137. Cambridge University Press, 1995.

[BW99] A. Burns and A. J. Wellings. How to verify concurrent Ada programs: the application of model checking. *ACM SIGADA Ada Letters*, 19(2):78–83, 1999.

[BWB+00] A. Burns, A. J. Wellings, F. Burns, A. M. Koelmans, M. Koutny, A. Romanovsky, and A. Yakovlev. Towards modelling and verification of concurrent ada programs using petri nets. In Pezzé, M. and Shatz, M., editors, *DAIMI PB: Workshop Proceedings Software Engineering and Petri Nets*, pages 115–134, 2000.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.

[Dil93] Laura K. Dillon. A visual execution model for ada tasking. *ACM Trans. Softw. Eng. Methodol.*, 2(4):311–345, 1993.

[Dil97] Laura K. Dillon. Task dependence and termination in ada. *ACM Trans. Softw. Eng. Methodol.*, 6(1):80–110, 1997.

[EKPPR03] S. Evangelista, C. Kaiser, J. F. Pradat-Peyre, and P. Rousseau. Quasar: a new tool for analysing concurrent programs. In *Reliable Software Technologies - Ada-Europe 2003*, volume 2655 of *LNCS*. Springer-Verlag, 2003.

[EKPPR04] S. Evangelista, C. Kaiser, J. F. Pradat-Peyre, and P. Rousseau. Verifying linear time temporal logic properties of concurrent ada programs with quasar. *Ada Lett.*, XXIV(1):17–24, 2004.

[Eva04] S. Evangelista. Helena, an efficient high level Petri nets analyser. Technical report, CEDRIC, CNAM, Paris, 2004.

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245. ACM Press, 2002.

[HRD04] John Hatcliff, Robby, and Matthew B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *Proceedings of the International Conference on Verification, Model Checking and Abstract Interpretation*, 2004.

[Int95] Intermetrics Inc. *Ada 95 Rationale*, 1995.

[MSS89] T. Murata, B. Shenker, and S.M. Shatz. Detection of Ada static deadlocks using Petri nets invariants. *IEEE Transactions on Software Engineering*, Vol. 15(No. 3):314–326, March 1989.

[NAC99] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. Data flow analysis for checking properties of concurrent java programs. In *Proceedings of the 21st international conference on Software engineering*, pages 399–410. IEEE Computer Society Press, 1999.

[NM94] M. Notomi and T. Murata. Hierarchical reachability graph of bounded Petri nets for concurrent-software analysis. *IEEE Transactions on Software Engineering*, Vol. 20(No. 5):325–336, May 1994.

[SMBT90] S.M. Shatz, K. Mai, C. Black, and S. Tu. Design and implementation of a petri net based toolkit for ada tasking analysis. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1(No. 4):424–441, 1990.

Appendix

```
1 procedure Dynamic_Eratho is
2
3    task type Prime is
4       entry Test_Primarity(Number : in Integer);
5    end Prime;
6
7    type Access_Prime is access Prime;
8
9    procedure Create_Prime(New_Prime : out Access_Prime) is
10   begin
11      New_Prime := new Prime;
12   end Create_Prime;
13
14   task body Prime is
15      My_Number    : Integer := 0;
16      Temp         : Integer := 0;
17      My_Next      : Access_Prime := null;
18      Termination  : Boolean := False;
19   begin
20      while not(Termination) loop
21         accept Test_Primarity(Number : in Integer) do
22            Temp := Number;
23         end Test_Primarity;
24         if (Temp = 0) then
25            -- Termination message
26            Termination := True;
27            if (My_Next /= null) then
28               -- Propagate the termination message
29               My_Next.Test_Primarity(0);
30            end if;
31         else
32            if (My_Number = 0) then
33               -- Store the prime number
34               My_Number := Temp;
35            else
36               -- Test the primarity
37               if ((Temp mod My_Number) /= 0) then
38                  if (My_Next = null) then
39                     -- If no neighbor, create one
40                     Create_Prime(My_Next);
41                  end if;
42                  -- Test the primarity on the neighbor
43                  My_Next.Test_Primarity(Temp);
44               end if;
45            end if;
46         end if;
47      end loop;
48   end Prime;
49
50   My_Prime : Prime;
51
52 begin
53   for Number in 2..5 loop
54      My_Prime.Test_Primarity(Number);
55   end loop;
56   -- Send the termination message
57   My_Prime.Test_Primarity(0);
58 end Dynamic_Eratho;
```

Fig. 15. Sieve of Eratosthene

```
1 procedure Server is
2    Max_Client : Integer := 5;
3
4    protected Data is                    -- The accessed data
5       procedure Get_Value (Value : out Integer);
6    private
7       Data_Value : Integer := 0;
8    end Data;
9
10   protected body Data is
11      procedure Get_Value (Value : out Integer)is
12      begin
13         Data_Value := Data_Value + 1;
14         Value := Data_Value;
15      end Get_Value;
16   end Data;
17
18   task type Thread is                  -- Only the threads can access the data
19      entry Get_Value (Param : out Integer);
20   end Thread;
21   type Access_Thread is access Thread;
22
23   task body Thread is
24   begin
25      accept Get_Value (Param : out Integer) do
26         Data.Get_Value(Param);
27      end Get_Value;
28   end Thread;
29
30   task type Task_Server is
31      -- The task server, create a thread for each client's request
32      entry Get_Thread(Id : out Access_Thread);
33   end Task_Server;
34
35   task body Task_Server is
36   begin
37      for I in 1..Max_Client loop
38         accept Get_Thread (Id : out Access_Thread) do
39            Id := new Thread;
40         end Get_Thread;
41      end loop;
42   end Task_Server;
43
44   The_Task_Server : Task_Server;      -- The task server
45
46   task type Client;                    -- Client of the task server
47   type Access_Client is access Client;
48
49   task body Client is
50      Id    : Access_Thread;
51      Value : Integer;
52   begin
53      The_Task_Server.Get_Thread(Id); -- Get the thread
54      Id.Get_Value(Value);             -- Get the value of the data
55   end;
56
57   A_Client : Access_Client;
58
59 begin
60   for I in 1..Max_Client loop         -- main loop, creates clients
61      A_Client := new Client;
62   end loop;
63 end Server;
```

Fig. 16. A client/Server program

```
 1 procedure Identification is
 2
 3    task type T3;
 4    task body T3 is
 5    begin
 6       null;
 7    end T3;
 8
 9    type Access_T3 is access T3;
10
11    task type T1;
12    task body T1 is
13       O : Access_T3;
14    begin
15       O := new T3;
16    end T1;
17
18    task type T2;
19    task body T2 is
20       P : Access_T3;
21    begin
22       P := new T3;
23    end T2;
24
25    T1_1 : T1;
26    T1_2 : T1;
27    T2_1 : T2;
28
29 begin
30    null;
31 end Identification;
```

Fig. 17. A basic Ada program.