A domain-specific language for task handlers generation, applying discrete controller synthesis

Gwenaël Delaval
INRIA Rhône-Alpes, POP ART
38330 Montbonnot, FRANCE
Gwenael.Delaval@inrialpes.fr

Éric Rutten
INRIA Futurs / LIFL, DaRT / WEST
USTL, 59655 Villeneuve d'Ascq, FRANCE
Eric.Rutten@inria.fr

ABSTRACT

We propose a simple programming language, called Nemo, specific to the domain of multi-task real-time embedded systems, such as in robotic, automotive or avionics systems. It can be used to specify a set of resources with usage constraints, a set of tasks that consume them according to various modes, and applications sequencing the tasks. We obtain automatically an application-specific task handler that correctly manages the constraints (if there exists one), through a compilation-like process including a phase of discrete controller synthesis. This way, this formal technique contributes to the safety of the designed systems, while being encapsulated in a tool that makes it usable by end-users and application experts. Our approach is based on the synchronous modelling techniques, languages and tools.

Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based Systems**]: Real-time and embedded systems; D.3.2 [**Programming Languages**]: Language Classifications—*Specialized application languages*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering*

Keywords

Real-time systems, safe design, domain-specific language, discrete control synthesis, synchronous programming

1. CONTEXT AND MOTIVATION

Embedded control systems are executing automatic control laws or signal processing, such as in robotic, automotive or avionics systems, or portable devices processing voice and image signal. They are reactive, in close interaction with their environment, including the controlled process, which has its own dynamics (following the laws of physics), imposing real-time management. The global behavior of such control systems results from this strong interaction. They are typically designed in terms of continuous models, and then implemented in a discretized form, as a *cyclic* computation upon sensor input data, producing extracted information, or control values towards actuators. This combination of computations and resource usage (sensors, processors, memory, power, actuators) defines a level of abstraction which we call a *task*. For a complex system, with different resources and a variety of functionalities, several control *modes* or *phases* can be designed, and switching between them has to be handled and controlled properly. This is intricate and the risk of errors is important, because of the complexity of systems and of requirements, particularly with respect to constraints on resource usage and interaction with the environment.

The decomposition we consider here is quite standard. Instances of systems structured this way can be found in robotics [3] or in StateFlow of Matlab/Simulink. Ways of integrating such heterogeneous formalisms have been proposed, e.g., in the concurrent modelling and design environment Ptolemy II [9], or at the modelling level in UML2 [6]. We position ourselves within this broad approach, by choosing the synchronous approach to reactive systems. In this approach, programming languages and commercial tools for such purposes usually combine data-flow and sequencing [10, 7]. Our work is based on this technical context, where task handlers can be seen as property-enforcing layers [1]. Our contribution is in proposing a method applying automated safe design techniques for the generation of task controllers.

Formal methods are a way to design safety-critical systems with an explicit care for their validation. A common practice consists of building up a specification, and then using *formal verification* (e.g., model checking) to assess properties. When a bug is detected, the designer uses a diagnosis to go back to the design and modify it, before verifying again. Such techniques are considered difficult to use because of the competence required in formal techniques. Much effort is devoted to *make them more user-friendly*, because they are to be applied by engineers specialists of the systems under design. A general notion of *invisible formal methods* advocates for fully automated techniques, integrated into design processes and tools. Some *programming languages* have compilers integrating verification, e.g., the synchronous languages [2] check for static properties. Explorations of dynamical behaviors in the reachable state space, integrated in the compilation, are less common [4]. We propose to integrate such a formal technique in a compilation-like tool.

Discrete controller synthesis consists of, given a property given as objective, computing the constraint (i.e., the controller) on transitions, if there exists one, such that the re-

SAC'06 April 23-27, 2006, Dijon, France

sulting constrained (i.e., *controlled*) behavior satisfies the property. It differs from verification in that it is more constructive, and proposes a solution. It has been studied in the synchronous framework [11]. It has been applied to the *modelling and control of multi-task systems* [12], where the set of tasks is modelled as a transition system, and a controller has to preserve constraints regarding the resources and the tasks sequencing. It can then be seen as the automatic generation of a *property-enforcing layer* [1] for a given system, or of an *application-specific scheduler* [8]. Our contribution here is to adapt such models and techniques to control tasks handlers, encapsulated into a simple domain-specific language.

We propose a *domain-specific language*, called Nemo [5], encapsulating controller synthesis for generating correct task managers. Its constructs describe domain-specific notions of resources and their constraints, tasks and their control, particular ordering constraints to be enforced, and applications built upon them. It is defined in terms of transition systems, temporal properties, and synthesis objectives. We produce *automatically*, through a compilation-like process including a phase of discrete controller synthesis, a *correct application-specific task handler* that satisfies the constraints (if there exists one). This way, this formal technique contributes to the safety of the designed systems, while being encapsulated in a tool that makes it usable by programmers. We use synchronous languages, modelling techniques and tools, and particularly the Mode Automata language [10] and the Sigali synthesis tool [11]. Our contribution lies in the proposal of this language, and the "invisible", encapsulated use of discrete controller synthesis.

2. DOMAIN SPECIFICITIES

2.1 Tasks, applications, resources

The *computation layer* performs data transformation algorithms, e.g., numerical computations implemented as C code, in an infinite loop. These computations, as shown in the lower part of figure 1, have basic control points. They can be *started*, which can involve initializations. They can *signal that they have reached their end*, i.e., that they are ready to stop (e.g., a control law has reached its objective within a given precision range: it may yet not stop but continue controlling the actuator around the objective). Some of then can be *stopped*, i.e., interrupted. They can be *suspended*, until they are resumed.

The *control layer* manages these computations' starts and stops, by encapsulating them into *tasks*, each provided with a local controller. As shown in the middle part of figure 1, this controller makes the relation between requests and starts, and between ends and stops: as will be discussed below, several variants may make sense. A task can also involve several *activity modes*, where the computation is different, as well as the resources engaged. These tasks can then be composed into *applications*, using structures such as loops, sequences, or parallel statements. We also consider alternative or choice statements, which are particularly meaningful here as will be seen further. As shown in the upper part of figure 1, an application can be requested and eventually stop, and they in turn send requests to underlying tasks. This whole *control layer* is a discrete event system, seen here as a synchronous reactive system [2].

Tasks use *resources*, for their computation (processors, memories, communication links), or in the embedding system (sensors and actuators). These resources involve *constraints*, such as: bounds on the number of simultaneous users, bounds on the available capacity, or the requirement to be always under control. Within a task, *modes* correspond to different resource consumptions, e.g., trade-offs between time, quality level (degraded modes) and memory or power consumption. The application sequences tasks, through a controller interacting with the tasks controllers, and the whole has to *preserve* the properties of the resources.

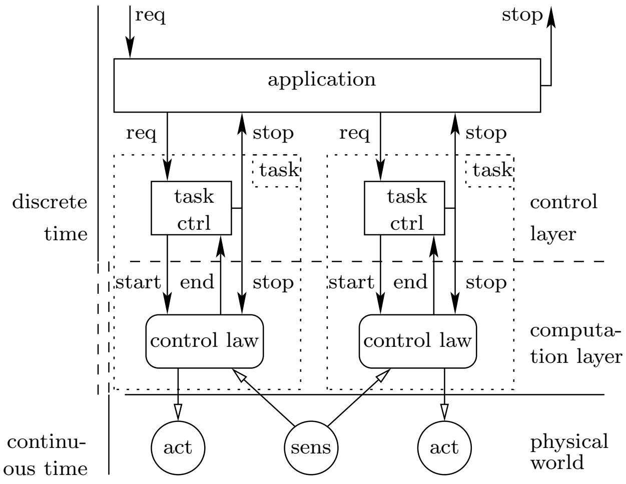


Figure 1: Control system

2.2 The controllable points

Control of a computation's termination relates *ends* (termination condition) and *stops* (actual termination).

Stop coming before end: some computations may be stopped without having yet reached their complete termination: e.g., anytime algorithms, where intermediary results can be delivered with intermediary quality. Such tasks can hence be interrupted: their *stop can be triggered.*

Stop coming after end: Some computations reach their objective, and then can continue cyclically in order to maintain it: an example is a robotic control law, giving the correction to be applied by actuators in order to near the objective. When it is reached, continuing will maintain the situation. This can be necessary: for example, in Orccad[3], the Robot-Tasks encapsulating a control law have a "transition phase" when the task is finished, but the next task isn't yet started: the task executes a "degraded mode" until the start of the next task, thus allowing the operation of actuators that must be always under control. Such tasks can be sustained beyond their end, which can be either *rejected*, i.e., the stop will occur at a later occurrence of the end, or *delayed*: the end is memorized, and the stop occurs later.

Control of the beginning relates *requests* and *starts.*

Start coming after request: when a request is made for a task, it might not be started, typically because of a resource not being available yet. The *request* can be either *rejected*, i.e., the start will occur at a later occurrence of the request, or *delayed*: i.e. memorized for the start to occur later.

Start coming before request, i.e., without an explicit request being made, e.g., default tasks for an actuator that must always be under control: their *start* can be *triggered.*

Controlling the modes during a computation involves switching between them. Modes are different ways to achieve the functionality of a task, which vary in the resources they consume, the time they take, the quality of service they achieve, or the energy consumption. For example, a task can be executed on one of two processors P_1 or P_2. We can say that this task is composed of two modes, each corresponding to one processor. Another example is a computation that can be performed by several algorithms, each of them using different amounts of the available resources.

By switching mode, one can for example make some available space in a bounded resource for other tasks to be able to start, or switch to a better quality and more costly mode, or unlock a task waiting upon an exclusive resource.

Controlling the choice in an alternative in applications is also a controllable point. Such a statement describes a choice between several branches: only one of them will be taken and executed. Each branch describes a sequence of tasks: the choice is made w.r.t. the particular sequences proposed, and considering the other tasks which can be in parallel in the complete application. The choice can exclude branches leading, along the sequence, to exclusive resource access conflicts, or too costly global consumptions.

3. OVERVIEW OF THE LANGUAGE

Due to space limitations, we present a rapid overview of the language, the complete description and formal definition being detailed in [5]. The formal framework involves transition systems, with weight functions on states, synchronous composition, and application of controller synthesis [1].

3.1 Programming task managers with Nemo

Nemo is devoted to build control layers. It allows for describing an abstraction of the computation layer, i.e., the *resources* used, the ways computations can be controlled, i.e., *tasks*, and some *explicit temporal properties* between *tasks*. These declarations are used to specify an *application*, as an ordering of tasks. From this, two basic elements are derived, as shown in figure 2: a *declarative part*, gathering constraints induced from resources, from explicit properties, and from the declared consumptions of tasks; and an *imperative part*, gathering observers for the explicit properties, behaviors of tasks and applications. These two parts constitute an incomplete specification. The imperative part specifies behaviors *a priori* not satisfying the constraints in the declarative part, but features points to be controlled. Therefore, obtaining the complete controller, satisfying these properties, involves applying some *resolution*. This requires the use of formal techniques, in order for the process to be automated, and encapsulated into a compiler-like tool. In our approach, the models are *automata*, and the resolution is *discrete controller synthesis*.

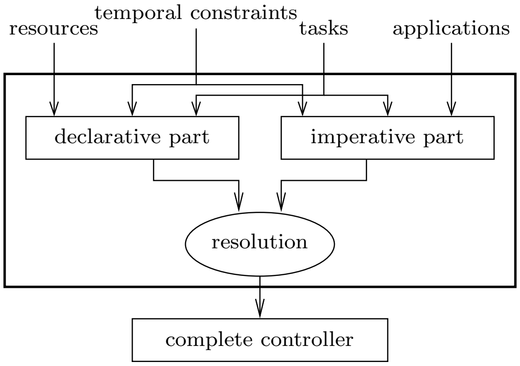
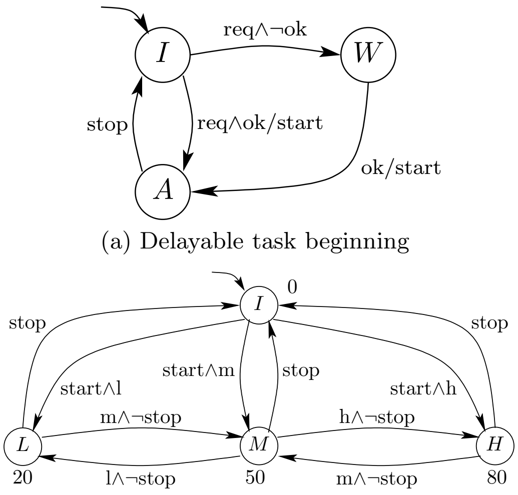


Figure 2: Compilation of Nemo.

3.2 The Nemo constructs

Figure 5 illustrates the constructs briefly introduced here.

Tasks are declared with optional properties of their activity, w.r.t. beginning and ending (triggerable, delayable or rejectable), and suspension. Then resources used are declared, with quantities, and whether they are used even when the task is suspended. Sets of modes are defined as a list of modes, each with the resources used, and the allowed transitions (see, e.g., the task `compute_move` in the Nemo program figure 5). Tasks are translated into a behavioural model in terms of automata. States represent the control. Transitions are labeled by events like requests and ends, and also by controllables like `ok` in figure 3(a), which shows the automaton for a delayable beginning. Use of resources is encoded as a weight function on the active state, or modes. For each task, such an automaton is composed in parallel with others concerning modes (figure 3(b), with weights 20, 50 and 80, where `l`, `m` and `h` are controllables) or suspension.



(a) Delayable task beginning

(b) Model of three modes of a task

Figure 3: Examples of tasks models

Resources and their implicit constraints are declared with optional properties like bounded number of users (e.g., exclusive control of the actuator `actuator_arm` in the example figure 5), bounded capacity (as in, e.g., a memory or CPU), or the need to be controlled by *at least one task* (e.g., actuators in a robotic system). These declarations will translate into properties like exclusivity of activity of tasks using the same resource, or bound on the value of the weight function corresponding to that resource, with addition of each task's consumption. These properties will be used as *controller synthesis objectives*.

Temporal constraints allow for explicit sequencing constraints to be specified, like `always t1 between (t2,t3)` and `always t1 before t2` for example; they can be composed using `or` and `and`, but not `not` in order to obtain safety properties. They are translated into observer automata, composed in parallel with the rest, and synthesis objectives excluding “bad” behaviors.

Applications are the imperative part of Nemo. Their purpose is the expression of an order of execution of the tasks, for a functionality to be performed. They define an intermediate layer emitting requests to task controllers, and waiting for ends, as shown in figure 1. Quite classical constructs are calling a task or application, sequence (`;`), parallel composition (`||`, starting simultaneously several branches), loop (repeating a sub-application until an event occurs), trigger (*s* `triggers` *app*, sending a request upon an input event). More originally, alternative (`|`) states that the application can be performed following either of the branches. The choice is left free, for the controller to decide, either at run-time, if both are potentially possible, or off-line, if the preservation of properties excludes one of them. An example of application can be seen in section 5, and a full definition is available in [5].

Applications are translated into incomplete automata representing the control structure. The translation of one statement is performed in a structural way from its composing statements. Each one is represented by an incomplete automaton, with a set of signals D to be emitted to start the statement (e.g., tasks requests for atomic statements), and one signal g reporting the end of the statement (e.g., the `stop` signals reporting tasks' ends). The incomplete automaton produced from one statement is then encapsulated into an automaton emitting the signals D at application's requests. Figure 4 shows the automaton for the sequence “$\texttt{app}_1$ `;` $\texttt{app}_2$”; the sub-automata $\texttt{app}_1$ and $\texttt{app}_2$ are the incomplete automata obtained from analysis of the two statements composing the sequence.

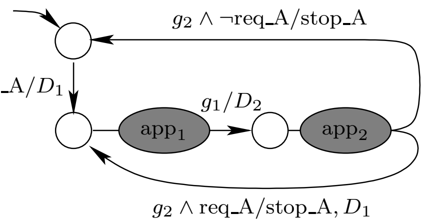

$g_2 \wedge \neg$req_A/stop_A
req_A/D_1
g_1/D_2
app$_1$
app$_2$
$g_2 \wedge$ req_A/stop_A, D_1

Figure 4: Automaton for “$\texttt{app}_1$; $\texttt{app}_2$”.

4. COMPILATION OF NEMO

Nemo is implemented as a concretization of figure 2. The compiler takes a Nemo program, and produces, accordingly to each entity (resource, temporal constraint, task, application), the behavioral parts in terms of Mode Automata, and the declarative parts, weights and objectives, in terms of boolean equations for the Sigali tool[1] [11]. The global automaton is compiled using Matou[2] [10], into an equational representation, for synthesis purposes, and into an executable format for execution and simulation. Discrete controller synthesis is performed by Sigali on the basis of the transition system and the objectives. The resulting controller is fed to a resolver, encapsulated into the interactive tool Sigalsimu, which performs a *co-simulation* of the executable representation with the controller.

Our approach strongly depends on the performance of the synthesis tool used. Indeed, the compilation of the language to automata and synthesis objectives is straigthforward. The actual bottleneck is the discrete controller synthesis, the complexity of which being of the same order as for model-checking verification. In order to evaluate the pertinence of the approach, we have measured the synthesis time on models composed of n start-rejectable tasks, for several values of n.

Nb of tasks	1	10	15	20	23
Synth. time	0.01s	0.34s	4.09s	4.74s	4.80s

Nb of tasks	24	26	28	30	32
Synth. time	54min.	10.79s	¿24h	24.21s	≈19h

The actual experimental results show that our approach is relevant for the studied domain, i.e. control systems. The synthesis tool has indeed been able to handle in a few seconds systems composed of more than 30 tasks. Nevertheless, one can notice several severe losses of performance, e.g. for systems composed of 24, 28 or 32 tasks. The synthesis time depends on many parameters, several of them being difficult to master, as, e.g., the order of the variables. This problem is a recurrent one in the framework of symbolic computation using BDDs; discussing it further is beyond the scope of this paper, but our work can benefit from results of the very active ongoing research in this area.

Therefore, we claim that the size of manageable systems is already meaningful, and rising, just like for model-checking verification. Furthermore, a particularly noteworthy point is that our approach comes as a subsitute to the traditional design/verification/correction one, which would be at least as expensive. In our approach, it is still possible to make errors in the models, or to build programs for which there is no solution: then you have to debug and re-design. But the point of comparison is that classically the controller would be designed “brainually”, i.e., by hours, or days, of hand and brain work. In our approach, part of it is generated automatically, by the synthesis of a correct solution.

5. TYPICAL EXAMPLE

The example in figure 5 gathers interesting features of Nemo. The system modelled is a robotic arm aimed at moving objects from a place to another. The two resources modelled are the actuator (the moving arm), and a CPU. The actuator is obviously an exclusive resource, and we also want its continuous control. The elements of the CPU is viewed, for this illustrative example, as the percentage of use of it. The system encloses two categories of tasks: `hold_arm` (“default” task holding the arm in its current position), `grab`, `move` and `release` actually manipulate the arm, whereas `check` and `compute_move` are computing tasks. We assume that the objects are brought toward the arm, e.g. by a conveyor belt. Some objects are delicate, so at any time, the operator can trigger a checking computation to insure that the system is currently able to handle such objects properly. The event `end_checking` signals that from now, all objects are standard and hence, the check computation won't need to be performed any more.

The synthesis computes a controller that will enforce the activation of the `hold_arm` task every time none of the other manipulating tasks is executed. It will also force the second branch of the alternative of `move_object`, as long as the task

[1] `http://www.irisa.fr/vertecs/Logiciels/sigali.html`
[2] `http://www-verimag.imag.fr/~maraninx/MATOU/`

```
resource actuator_arm:
    exclusive;
    steady control;
end resource;
resource CPU:
    composed of 100 elements;
end resource;
task hold_arm:
    start triggerable,rejectable;
    stop triggerable,rejectable;
    uses actuator_arm, 5 of CPU;
end task;
task grab:
    uses actuator_arm, 20 of CPU;
end task;
task move:
    uses actuator_arm, 20 of CPU;
end task;
task release:
    uses actuator_arm, 20 of CPU;
end task;
task check:
    uses 40 of CPU;
end task;
task compute_move:
  modes
    Precise_move : uses 80 of CPU;
    Rough_move : uses 50 of CPU;
    trans Precise_move <-> Rough_move;
  end modes;
end task;
property
  always move between (grab,release)
end property;
application move_object:
   (grab;(compute_move||move);release)
 | (grab;compute_move;move;release)
end application;
application main:
 loop (move;move_object) until end_app ||
 loop (sig_check triggers check)
 until end_checking
end application;
```

Figure 5: Nemo example

`check` can be triggered, because the CPU would not have the capacity to handle the three tasks `compute_move`, `move` and `check` at the same time. This does show the interest of using discrete controller synthesis, as the choice involves a look ahead in all the possible paths for a specific application. One can also notice that the explicit property "always move between grab and release" won't need any controller synthesis, as it is enforced by the application. In this case, the controller synthesis computation behaves as a verification: if the property is satisfied by the transition system, the controller gives no additional constraint. If it hadn't been the case, in another application, the synthesis would have failed, because the tasks do not have enough controllability.

6. CONCLUSION AND PERSPECTIVES

We presented a *domain-specific language*, devoted to task handlers, and its *implementation*. Its definition involves a model of tasks control as transition systems, and the application of *discrete controller synthesis techniques*. The framework is an instance of user-friendly, "invisible" formal method, where the final user need not know about the underlying technicalities. The approach is adapted to the application area of embedded control systems. It shares concepts with existing tools of this area, such as Orccad [3]; the addition is in the automated generation of task controllers. The performances of the underlying controller synthesis tool [11] gives good hopes for upscaled applications.

Perspectives are in different directions. Regarding useability, a diagnosis should be given in case there is no solution to the synthesis. The integration of the generated controller into a run-time executive can be seen as interfacing a component amongst the others. Our model of tasks is currently quite simplified: it could be refined with e.g., sequential phases, interrupts, fault-tolerance (e.g., model of the environment, fault model), more elaborate models of architectures and power consumption. The discrete controller synthesis is costly, but it can be improved in the encodings of the models themselves, which have not been optimized in this first version. It can be noted that regarding the algorithms, which are out of our scope, progress is going the same way as for model-checking, which is making ever larger models amenable to the technique. We currently use only invariance objectives for synthesis: we intend considering attractivity, or quantitative optimization along paths. We also want to integrate some alternatives to synthesis, like managing some of the constraints with coordination code.

7. REFERENCES

[1] K. Altisen, A. Clodic, F. Maraninchi, and E. Rutten. Using controller synthesis to build property-enforcing layers. In *European Symposium on Programming (ESOP)*, Warsaw (Poland), Apr. 2003. LNCS 2618.

[2] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages twelve years later. *Proc. of the IEEE, Special issue on embedded systems*, 91(1):64–83, Jan. 2003.

[3] J.-J. Borrelly, E. Coste-Manière, B. Espiau, K. Kapellos, R. Pissard-Gibollet, D. Simon, and N. Turro. The Orccad architecture. *Int. J. of Robotics Research*, 17(4), 1998.

[4] A. Chakrabarti, L. de Alfaro, T. Henzinger, and F. Mang. Synchronous and bidirectional component interfaces. In *CAV 02: Int. Conf. on Computer Aided Verification*, 2002.

[5] G. Delaval and E. Rutten. A domain-specific language for multi-task systems, applying discrete controller synthesis. Research Report 5690, INRIA, Sept. 2005. `http://www.inria.fr/rrrt/rr-5690.html`.

[6] B. Douglass. *Real Time UML*. Addison Wesley, 2004.

[7] Esterel technologies. The Esterel Studio and Scade tools. `http://www.esterel-technologies.com`, 2005.

[8] C. Kloukinas and S. Yovine. Synthesis of safe, qos extendible, application specific schedulers for heterogeneous real-time systems. In *15th Euromicro Conference on Real-Time Systems (ECRTS'03)*, Porto, Portugal, 2003.

[9] E. Lee. The Ptolemy project. University of Berkeley. `http://ptolemy.eecs.berkeley.edu`, 2005.

[10] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Sc. of Computer Programming*, (46):219–254, 2003.

[11] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the signal environment. *Discrete Event Dynamic System: Theory and Applications*, 10(4), October 2000.

[12] H. Marchand and E. Rutten. Managing multi-mode tasks with time cost and quality levels using optimal discrete control synthesis. In *14th Euromicro Conference on Real-Time Systems (ECRTS'02)*, June 2002.