Aspects of Availability

Enforcing timed properties to prevent denial of service

Pascal Fradet

INRIA
Inria Grenoble Rhône-Alpes, 655, av. de l'Europe, 38330 Montbonnot, France

Stéphane Hong Tuan Ha

INRIA / CEA Saclay
CEA Saclay, DRT/LIST/DTSI/LSL, 91191 Gif sur Yvette Cedex, France

Abstract

We propose a domain-specific aspect language to prevent denial of service caused by resource management. Our aspects specify availability policies by enforcing time limits in the allocation of resources. In our language, aspects can be seen as formal timed properties on execution traces. Programs and aspects are specified as timed automata and the weaving process as an automata product. The benefit of this formal approach is two-fold: the user keeps the semantic impact of weaving under control and (s)he can use a model-checker to optimize the woven program and verify availability properties. This article presents the main approach (programs, aspects, weaving) formally using timed safety automata. The specification of resources, optimizations and verification are sketched in a more intuitive fashion. Even if a concrete implementation remains as future work, we address some high-level implementation issues and illustrate the approach by small examples and a case study.

Key words: Aspect-Oriented Programming, Availability, Resource Management, Timed Automata, Weaving, Denial of Service

1 Introduction

Along with confidentiality and integrity, *availability* is one of the three main classes of security properties. Availability guarantees that the requests of au-

Email addresses: Pascal.Fradet@inria.fr (Pascal Fradet), Stephane.Hong-Tuan-Ha@cea.fr (Stéphane Hong Tuan Ha).

thorized subjects are satisfied in a timely manner. In other words, there is no *denial of service*. Like the other security properties, the implementation of availability crosscuts the basic functionality of programs and produces tangled code. In this paper, we use aspect-oriented techniques to express resource management and address the prevention of denial of service (*i.e.*, availability) separately from the basic functionality. That separation of concerns leads to programs that are easier to develop and maintain. This is especially useful in a security context where programs may have to be changed quickly to respond to new threats.

We propose a domain-specific aspect language in order to prevent denial of service caused by resource management (*e.g.*, starvation, deadlocks, etc.). Aspects specify availability policies which enforce time constraints on resource allocation. For example, a constraint may be that a service `S` does not retain a resource `R` more than k seconds or that it does not allocate the resource `R2` less than k seconds after it has released `R1`. To the best of our knowledge, this is the first work using aspects to enforce the availability of resources.

In our language, an aspect can be seen as a timed property on execution traces which specifies an availability policy. The semantics of base programs and aspects are expressed as *timed automata* [1]. The automaton representing a program specifies a superset of all possible (timed) execution traces whereas the automaton representing an aspect specifies a set of desired/allowed (timed) execution traces. Weaving can be seen as a *product* of two timed automata (*i.e.*, the intersection of execution traces) which restricts the execution of the base program to the behaviors allowed by the aspect.

In general purpose languages, aspects are often described in a syntactic fashion as directives of code insertion at explicit join points. Such a code is not restricted and, consequently, can completely distort the semantics of the base program. In contrast, our aspects are constrained and have a more semantic nature: they specify sets of desired timed behaviors. The main advantage of such a formal approach is two-fold:

- aspects are expressed at a higher-level and the semantic impact of weaving is kept under control;
- model checking tools (*e.g.*, UPPAAL [2,3]) can be used to optimize weaving and verify the enforcement of general availability properties.

Section 2 outlines our framework, in particular: the systems and availability properties considered, the general approach and a small example used throughout the paper to illustrate the different steps. We briefly recall the main characteristics of timed automata in Section 3. The specification of resources, illustrated with two standard types of resources, is described in Section 4. Sections 5 and 6 present the syntax and semantics of services and

availability aspects, respectively. The technical core of the paper lies in Section 7 which describes the abstraction of services and semantics of aspects in terms of timed automata and the weaving as an automata product. Section 8 sketches the optimization, verification and concretization of the final (woven) automaton back into a source program. Even if a concrete implementation remains as future work, we address in Section 9 some key implementation issues. Section 10 presents a case study where several temporal constraints of an automatic teller machine are implemented as aspects. We conclude by presenting related work in Section 11 and possible extensions in Section 12.

This article extends and revises the work presented in GPCE'07 [4]. Sections 4, 9, 10, and 11 are new. We have clarified and/or simplified some technical points (*e.g.,* representations and translations) and have added more explanations and examples. Older, preliminary versions have also been published in a French conference [5], journal [6] and PhD thesis [7]. Correctness proofs of our approach in a simpler setting can be found in [7].

2 Framework

We first define the systems and availability problems considered. Then, we present our approach and the example used thereafter to illustrate it.

2.1 Systems and availability

We consider systems that can be decomposed along three layers: *users*, *services* and *resources* (Figure 1). Users send their requests to services and wait for the answer. Services process users' requests sequentially. Requests are stored in a FIFO queue; processing a request involves computation and accesses to resources. Resources are (logical or physical) entities shared among services. For instance, files, printers, processors or network connection managers are examples of resources. This type of client-server model is of widespread use in web servers and distributed applications. We suppose that the numbers of services and resources are fixed and known.

Each service can be seen as a non-terminating loop processing requests: the request is fetched, processed, the result is sent to the corresponding user and so on. We do not specify users and how services deal with user requests any further. Since we are interested in resource management and the prevention of denial of service, we focus on interactions between services and resources.

The availability problems we consider come from concurrent accesses of ser-

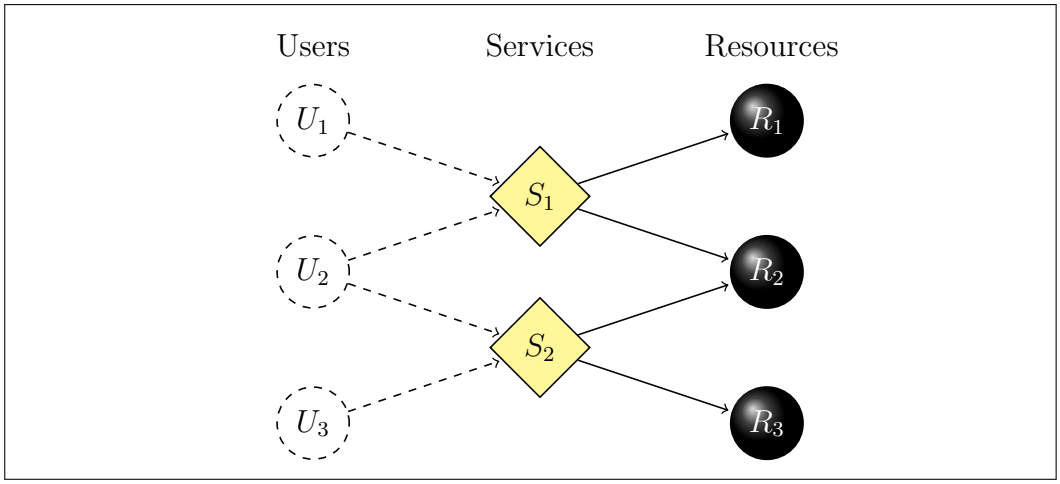

Fig. 1. Three-layer model

vices to shared resources. For instance, there can be starvation when a service cannot allocate a resource or deadlocks when two services wait for a resource allocated by the other one. Such problems can be prevented by appropriate resource management. Of course, hardware faults can also cause availability problems. This source of denial of service must be addressed by dedicated fault-tolerance techniques (see for example [8,9]).

2.2 Approach

Yu and Gligor have studied denial of service caused by resource management [10]. They have shown that availability properties depend not only on resources but also on constraining the behavior of services using user agreements. Our resource management system is inspired by Yu and Gligor's model. As illustrated in Figure 2, it is made of two parts:

- the specification of resources in terms of sufficiently precise automata which can be translated into programs. Several types of resources (exclusive access, shareable) have been specified in [5].
- the specification of constraints on the use of resources. We define these constraints as *availability aspects* which are woven on the source code of services. Compared to other aspects, availability aspects are original in that they specify timed behaviors. They can, for example, limit the amount of time a service may allocate a resource or forbid too frequent reallocations of a resource by the same service (see Section 6).

In this paper, we present informally the specification of resources and mostly focus on the aspect-oriented part of the framework. Resource management constraints are specified by an availability aspect per service. Each aspect is independent and defines a local policy which is woven on the corresponding

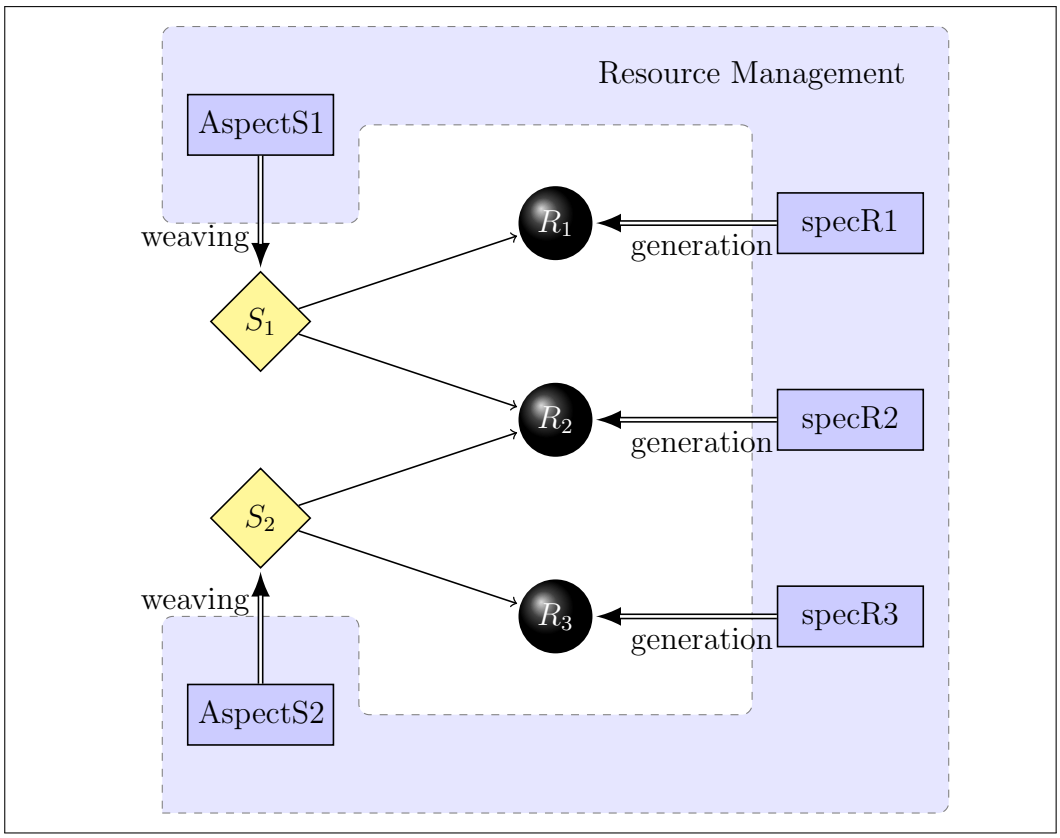

Fig. 2. Global layout of the system

service. These aspects correspond to Yu and Gligor's user agreements. We do not consider global aspects constraining services depending on the behavior of other services. They are more expressive but their implementation involves a global monitor observing the execution of the complete system. Local aspects are sufficiently expressive to prevent most denial of service and their implementation can be optimized using static weaving.

Our approach relies on timed automata and weaving, the key transformation step, is specified as a timed automata product. The technical core of our technique is made of the following steps:

- a service is abstracted into a timed automaton over-approximating its execution traces and its timed behavior (Section 7.1);
- an aspect is defined using a domain-specific language. Its semantics is given by a timed automaton (Section 7.2);
- the aspect is woven to the service by performing the product of the two corresponding automata. The product automaton represents a refined service that satisfies the constraints of the aspect (Section 7.3);
- information about the execution times of service instructions can be taken into account, again using automata product. This permits to optimize the woven automaton (Section 8.1);

- it is possible to automatically verify that the woven automaton satisfies general availability properties (Section 8.2);
- the last step amounts to concretizing the (optimized and verified) automaton into source code using timed commands (watchdog timers, waiting loops, interrupts) (Section 8.3).

2.3 System example

We will use the example of Figure 3 to illustrate the different steps of our technique. This small system is made of two resources (`M1` and `M2`) with exclusive access and two services (`S1` and `S2`) with a non terminating loop request. The

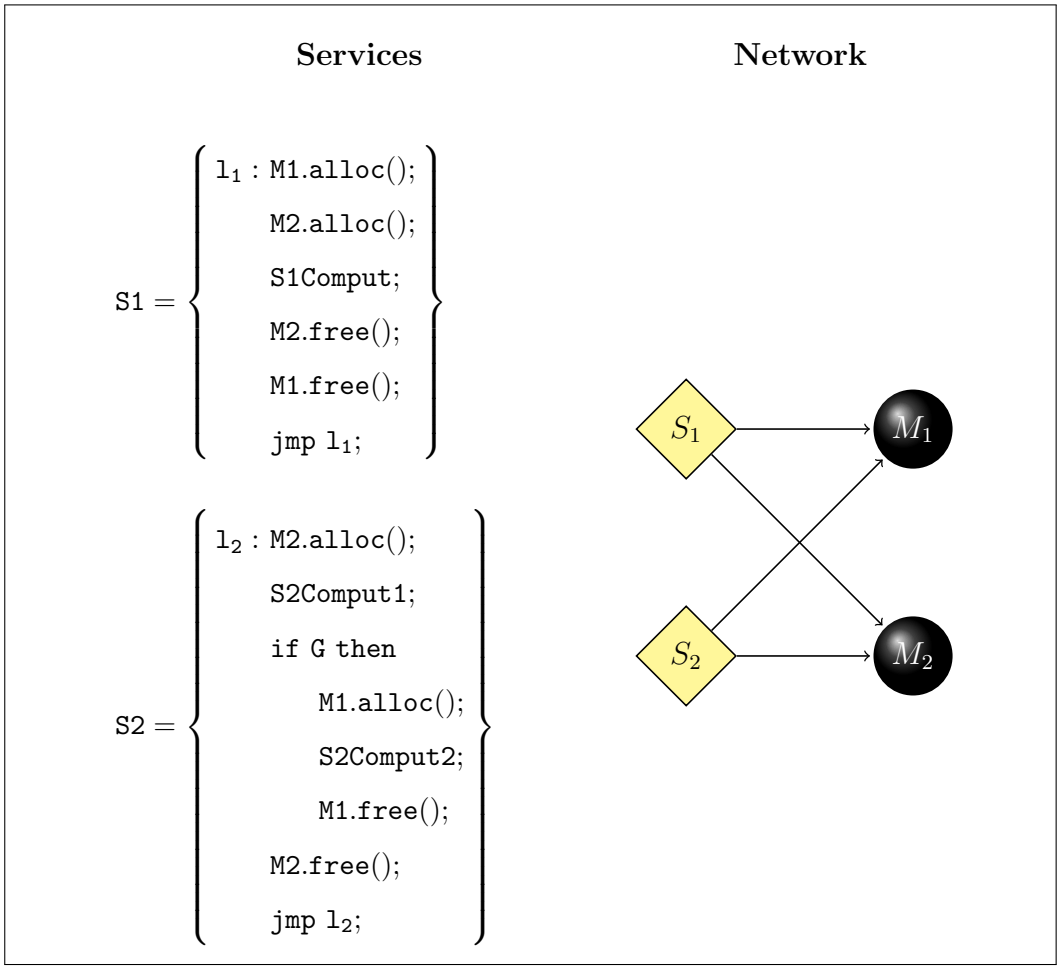

Fig. 3. A simple system with two services and two resources

service `S1` allocates the resource `M1` then `M2` (`M1.alloc(); M2.alloc();`). It computes `S1Comput` (which takes between 2 and 10 seconds), releases the resources `M2` and `M1` and iterates. The service `S2` models a potentially dangerous behavior. It allocates the resource `M2`, then computes `S2Comput1` which takes at least 1 second (and may not terminate). If the guard `G` is true, it allocates

M1, computes S2Comput2 (which takes between 3 and 20 seconds) and releases M1. It releases M2 and iterates.

The resource management of this system may lead to two availability problems:

- starvation may occur if S2Comput1 does not terminate. In this case, the service S2 never releases M2 which is needed by S1;
- deadlock may also occur when the service S1 has allocated the resource M1 and waits for M2 while the service S2 has allocated the resource M2 and waits for M1.

3 Timed automata

In this section, we briefly recall the syntax and semantics of timed automata which we use to model programs, aspects and weaving. Timed automata have been introduced to specify problems and to verify properties where time is explicit. We present *timed safety automata* [2,11] which are a commonly used kind of timed automata.

3.1 Syntax

Let H be a set of real valued variables used to represent clocks. A *clock constraint* C is of the form

$$C ::= x \odot k \ \mid \ x - y \odot k \text{ with } x, y \in H, \quad k \in \mathbb{N} \text{ and } \odot \in \{\leq, <, =, >, \geq\}$$

Transitions of timed automata are guarded by a set of clock constraints (to be interpreted as the conjunction of the constraints). We write 2^C for the set of possible guards (*i.e.*, clock constraints).

A timed automaton A is a tuple $(Q, q_0, H, \Sigma, \rightarrow_a, I)$ where:

- Q is a finite set of states;
- $q_0 \in Q$ is the initial state;
- H is a finite set of clocks;
- Σ is a finite set of labels denoting events/actions of the automaton;
- $\rightarrow_a \subseteq Q \times 2^C \times \Sigma \times 2^H \times Q$ is the transition relation;
- $I : Q \rightarrow 2^C$ maps a state to its *invariant*.

A transition $(q, g, a, r, q') \in \rightarrow_a$ specifies that the automaton can go from state q to state q' by performing the action a and resetting the set of clocks r ($r \in H$) if the guard g is true. The sub-set of clocks r is called a reset. We restrict an invariant to be a conjunction of constraints of the form $x \leq k$ or $x < k$ with k an integer.

The symbol . is overloaded to denote the empty guard (*i.e.*, $\emptyset$ or **true**), the empty reset (*i.e.*, $\emptyset$) and the empty action more commonly written ϵ. We also write $q \xrightarrow{g,a,r} q'$ for transitions; for example, $q \xrightarrow{\cdots} q'$ denotes the spontaneous transition.

3.2 Semantics

The operational semantics of a timed automaton $A = (Q, q_0, H, \Sigma, \rightarrow_a, I)$ is given by a transition system between states of the form (q, u) where $q \in Q$ is the current state of the automaton and the function $u : H \rightarrow \mathbb{R}$ maps clocks to their current value. The initial semantic state is made of the initial state of the automaton and the function returning 0 for all clocks.

The definition of the semantic transition relation makes use of the following notations. Let $u : H \rightarrow \mathbb{R}$ mapping clocks to their values, g a guard (*i.e.*, a set of clock constraints) and q a real number then:

- $u \in g$ denotes that clocks of u (H) verify the guard g;
- $u + d$ denotes that d is added to all clocks of u;
- $u[r \mapsto 0]$ denotes the reset of all clocks of the set r.

The semantic transitions are either transitions representing the time passing

$$(q, u) \rightarrow (q, u + d) \text{ if } \forall d' : 0 \leq d' \leq d \Rightarrow u + d' \in I(q)$$

or transitions representing the execution of an action

$$(q, u) \rightarrow (q', u') \text{ if } \exists q \xrightarrow{g,a,r} q' \text{ such that } u \in g,\ u' \in I(q'),\ u' = u[r \mapsto 0]$$

Time may pass only if it satisfies the invariant of the current state. A transition of the automaton may occur if and only if its guard and the invariant of the new state are satisfied. The semantics of the automaton is the set of traces of the associated transition system.

The first automaton in Figure 4 enforces that the action a is performed at least before 10 time units (initially or after each action a). The state invariant

prevents the automaton from waiting more than 10 time units before performing a. The clock x is initially set to 0 and is reset after each a. The second automaton enforces to wait at least 5 time units before performing an a. Each time the action a is performed, the clock y is reset and an a transition can only occur when $y \geq 5$. The clocks are assumed to be initialized to zero.

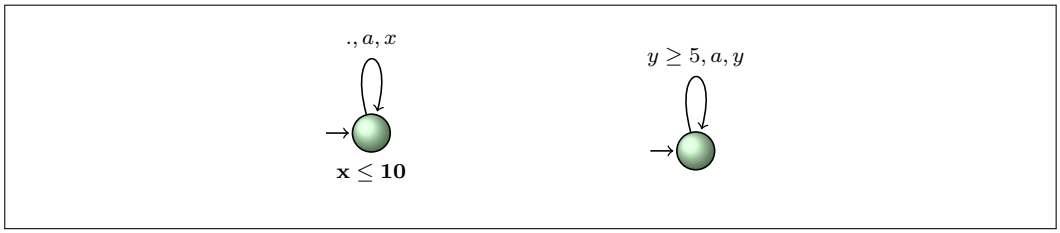

Fig. 4. Simple timed automata

3.3 Timed automata product

The product of two timed automata $X = (Q_x, x_0, H_x, \Sigma, \rightarrow_x, I_x)$ and $Y = (Q_y, y_0, H_y, \Sigma, \rightarrow_y, I_y)$ with the same set of actions and disjoint sets of clocks is the automaton $X \otimes Y = (Q_x \times Q_y, (x_0, y_0), H_x \cup H_y, \Sigma, \rightarrow, I)$ with:

$$I(x, y) = I_x(x) \cup I_y(y)$$

$$\textsc{Action} \quad \frac{x_1 \xrightarrow{g_x, a, r_x}_x x_2 \qquad y_1 \xrightarrow{g_y, a, r_y}_y y_2}{(x_1, y_1) \xrightarrow{g_x \cup g_y, a, r_x \cup r_y} (x_2, y_2)}$$

$$\epsilon_1 \quad \frac{x_1 \xrightarrow{g_x, ., r_x}_x x_2}{(x_1, y) \xrightarrow{g_x, ., r_x} (x_2, y)} \qquad \epsilon_2 \quad \frac{y_1 \xrightarrow{g_y, ., r_y}_y y_2}{(x, y_1) \xrightarrow{g_y, ., r_y} (x, y_2)}$$

The states of the product automaton is the cartesian product of the states of the two automata X and Y. The initial state is made of the initial states of X and Y. The invariant of a product state is the conjunction (union) of the invariants of its two constituent states.

The transition relation of the product automaton is defined by three rules. The rule Action denotes the case where an action is performed by both automata. The guard is the conjunction of the two constituent guards. It is expressed as a union ot the sets representing guards (recall that these sets are interpreted

as the conjunction of their elements). The set of clocks to reset is the union of the two reset sets. The rules ϵ_1 and ϵ_2 denote the cases where one of the two automata performs the empty action. In these cases, the automata can proceed independently.

The execution traces recognized by the product automaton $X \otimes Y$ is the intersection of the execution traces recognized by the two automata X and Y.

The product of the two automata of Figure 4 is represented in Figure 5. It specifies the intersection of the allowed traces of the two automata. It enforces that the delay between each action a lies between 5 and 10 time units.

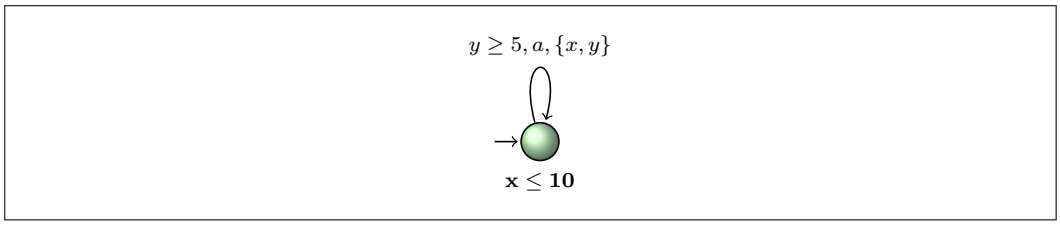

Fig. 5. Simple product automaton

4 Resources

Resources (communication, memory, CPU, *etc.*) play a central role in availability. Even if our main focus is the weaving of timed properties on services, we sketch in this section how resources can be specified in the same framework (*e.g.*, timed automata) using examples. In order to verify global availability properties, resources as the other components of the system (services and aspects) must be formally specified.

In our approach, resources are specified by:

- an *interface* listing all the instructions to access them;
- an *automaton* specifying their behavior, in particular:
  - the evolution of their internal state according to the different accesses made by services;
  - the management of the requests from services.

We use the timed automata of UPPAAL [2,3] to specify the behavior of resources. UPPAAL automata extend timed automata with urgent states (where time is not allowed to pass), synchronization communications, arrays and bounded integer variables. These extensions permit to represent programs and aspects more concisely. It is however only syntactic sugar which can be translated into pure timed automata by encoding (*e.g.*, adding new clocks to express

urgency) and state enumeration. Another benefit of using UPPAAL is to allow the verification of high-level availability properties of the woven automaton by model-checking (see Section 8.2).

If the UPPAAL automaton is deterministic and complete (all conditional cases are taken into account), it is easy to generate executable code from such a specification. We do not describe formally UPPAAL syntax and semantics. Instead, we explain the extensions intuitively as they are used in the examples. The reader will find an abundant documentation about UPPAAL (manual, tutorial, articles) at `http://www.uppaal.com/`.

The representation of a resource as an automaton is more declarative than a direct encoding as a source program. Further, since it is in the same formalism as services (base and woven), the global system can be described and analyzed using UPPAAL.

In this section, we focus on two common types of resources: resources with `exclusive` access and `sharable` resources.

4.1 `Exclusive` *access resources*

Exclusive access resources are used by a single user at a time. They are said to belong to the `mutex` type. That kind of resource protects access (reads and writes) of shared data. Different specifications can be considered.

A simple specification can be given by:

- an interface with only two operations:
  - `alloc()` that takes the resource (*i.e.*, enters the critical section);
  - `free()` that releases the resource (*i.e.*, exits the critical section).
- a specification of resource management where:
  - `alloc()` are performed by choosing randomly a requesting service when the resource is free;
  - `free()` are performed without delay.

More precisely the resource management is specified by the automaton of Figure 6.

Initially, the resource is free and is granted to a service requesting it. The synchronizing communications *alloc* and $free$ are supposed to be unique to a resource (*e.g.*, they can be indexed by the resource ID). A service accesses a resource using the synchronizing communications *alloc*? to request and take it and $free$! to release it. Time is allowed to pass when the service performs *alloc*? but not $free$!. In the latter case, the service must be in an urgent state

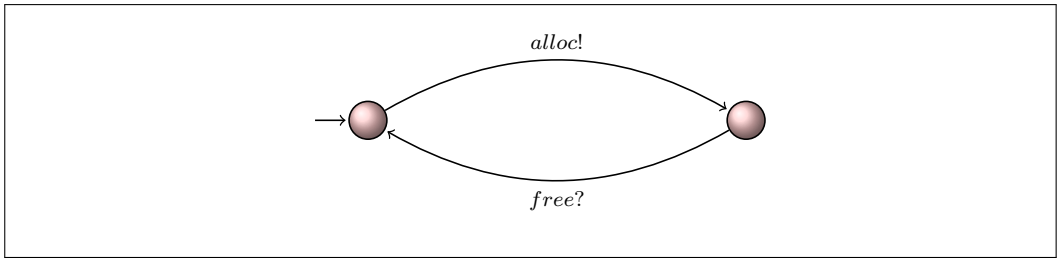

Fig. 6. Basic behavior of mutex resource

in order to release the resource without delay.

This simple description is correct but does not ensure fairness of the allocation. Starvation can arise since the resource is granted to a service chosen randomly among all services requesting the same resource. A more refined specification manages requests in a FIFO fashion. It is described by the automaton of Figure 7.

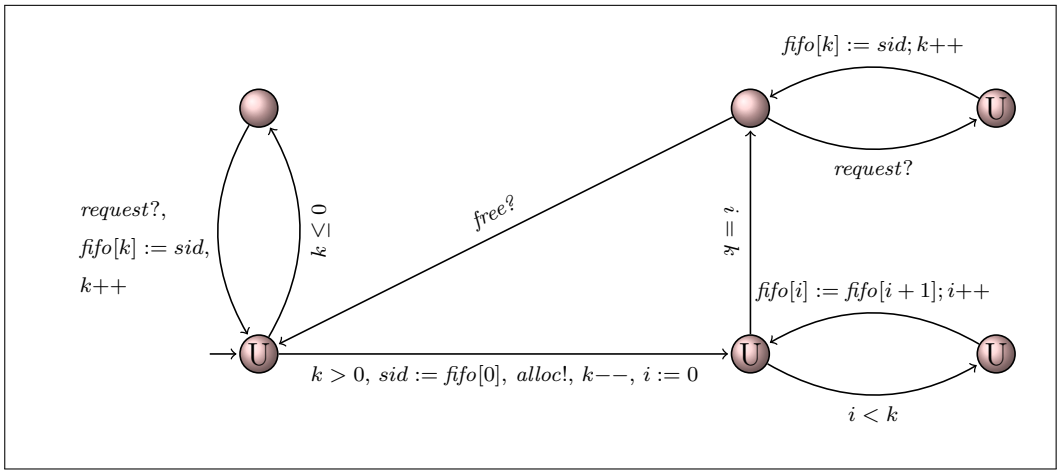

Fig. 7. FIFO management of mutex resource

The automaton uses the notion of urgent states (marked with a U) and two bounded integer variables $0 \leq k \leq max$ and $0 \leq i \leq max$ where max represents the maximum number of services. The automaton uses an array (*fifo* bounded by max) to implement the FIFO file storing requests of services. Initially, the file is empty ($k = 0$) and the resource waits for a request. The shared variable *sid* is the ID of the service performing the request. That ID is stored in *fifo* and the allocation is performed right away. Next, the array is updated (*i.e.*, the first element is removed and the others shifted) using the index i. The resource waits for the deallocation (*free*?) but, in the meantime, accepts incoming requests and stores into the array. After the deallocation, either the file is empty and the resource waits for another request, or the first request of the file is processed. Note that the only two states where time is allowed to pass are states where the resource waits for requests and deallocations.

That specification supposes that services request the resource using a transition of the form $request!, sid := myid$. That transition may be urgent since the resource is always ready to accept requests. When the synchronization takes place, the shared variable *sid* has the identity of the selected service. A service that has performed a request must wait for the allocation using a transition of the form $sid = myid, alloc!$. The guard $sid = myid$ ensures that the request currently processed is from the right service. Services may only release (*free*!) resources that have been allocated to them.

The automaton of Figure 7 is complete and deterministic. It is used to generate the following CSP-like code.

```
init : if k<=0 then request?; fifo[k]:=sid; k++;
       sid:=fifo[0]; alloc!; k--;
       i:=0;
       while i<k do fifo[i]:=fifo[i+1]; i++; od;
loop :   free?    -> jmp init;
      || request? -> fifo[k]:=sid; k++; jmp loop;
```

In that language, we suppose that communication instructions are blocking and when several guards are enabled the first one is chosen. In our example, if *free*? and *request*? are enabled then *free*? is chosen.

Such a specification of `mutex` resources ensures some basic availability (fairness) properties. For example, a service cannot repeatedly allocate the same resource when others are waiting for it. Still, it can suffer from several availability problems.

- if a service takes a resource and fails to release it;
- deadlocks are possible when several services share several `mutex` resources.

4.2 `Sharable` *resources*

Mutex resources can be refined into resources made of a collection of k parts which can be allocated to several services. Such shareable resources are common [12]: servers, memory and CPU can be seen as shareable (at least at a proper abstraction level). They are typically defined by:

- an interface made of three operations:
  - `request(i)` to request i parts of the resource;
  - `alloc(i)` to grant i parts;
  - `free(i)` to release i parts.

- the behavior, where usually:
  · `request(i)` are processed in a FIFO ordering;
  · `alloc(i)` are granted as soon as i parts are free;
  · `free(i)` are performed without delay.

We do not give the corresponding automaton which uses similar encodings as the automaton of Figure 7.

A shareable resource may cause the same availability problems as a `mutex` resource. Indeed, a shareable resource with k parts can be seen as k `mutex` resources. Therefore, contrary to `mutex` resources, deadlocks can arise with a single shareable resource. Shareable resources may profit from more complex availability policies *e.g.,* managing quotas on the number of parts allocated by services.

Many other kinds of resources or more sophisticated specifications can be described in this framework (*i.e.*, as UPPAAL automata). For instance, it would be possible to associate services with priorities. Requests would be processed depending on their ordering *and* the priority of the corresponding service.

5 Services

In this section, we describe the syntax and semantics of the source language of services.

5.1 Syntax

A service is defined by a set of *instructions* $\{I_1, \ldots, I_n\}$ of the form

$$I ::= l_1 : c \rightsquigarrow l_2 \mid l_1 : g \rightsquigarrow l_2 \ ; \ l_3$$

where l_1, l_2 and l_3 are *labels*, c a *command* (*e.g.,* an assignment) and g a *test* (*i.e.*, a boolean expression). In the following, we use *action* to denote either a command or a test. Intuitively, if the current program point is l_1 and the service S contains the instruction:

- $l_1 : c \rightsquigarrow l_2$ then the command c is performed and the current program point becomes l_2 ;
- $l_1 : g \rightsquigarrow l_2 \ ; \ l_3$ then if the test g is true the current program point becomes l_2 else it becomes l_3.

Left-hand side labels are supposed to label a unique instruction. This syntactic restriction ensures sequentiality and determinism of services (provided that commands are sequential and deterministic).

The source language is very simple. Its main advantage is that programs are very close to their control flow graph which will be translated to a timed automaton. A higher-level language could be considered by using a control flow analysis to abstract programs into automata.

Typically, a service is an infinite loop waiting for a user's request, processing and answering the request and so on. The loop of a service starts with the instruction $l_0 : \texttt{getUser}() \rightsquigarrow l_1$ which waits and takes a new request and ends with $l_i : \texttt{endUser}() \rightsquigarrow l_0$ which returns the results to the user and jumps to l_0 to treat a new request. For example, the service `S1` of Figure 3 can be written in that syntax:

$$\texttt{S1} = \left\{ \begin{array}{ll} l_0 : \texttt{getUser}() & \rightsquigarrow l_1 \\ l_1 : \texttt{M1.alloc}() & \rightsquigarrow l_2 \\ l_2 : \texttt{M2.alloc}() & \rightsquigarrow l_3 \\ l_3 : \texttt{S1.comput}() & \rightsquigarrow l_4 \\ l_4 : \texttt{M2.free}() & \rightsquigarrow l_5 \\ l_5 : \texttt{M1.free}() & \rightsquigarrow l_6 \\ l_6 : \texttt{endUser}() & \rightsquigarrow l_0 \end{array} \right\}$$

The commands `getUser`() and `alloc`() are blocking (*e.g.*, if there is no request or if the resource is not available); the command `S1.comput` denotes a potentially large collection of basic instructions without any resource management command.

5.2 Semantics

The semantics of a service S is expressed as a labeled transition system $(\Sigma_S, (l_0, s_0), \mathcal{E}_S, \longrightarrow_S)$ where:

- Σ_p is an infinite set of states (l, s) with l a label and s a store;
- (l_0, s_0) is the initial state;
- $\mathcal{E}_S$ is the set of actions of S;
- $\longrightarrow_S$ is the transition function labeled by the action.

The semantics of commands c is assumed to be given by the function $\mathcal{C}[\![c]\!]$ mapping the current store to the updated store. The semantics of tests is assumed to be given by the function $\mathcal{G}[\![g]\!]$ which takes the current store and returns a boolean. The transition function can be then defined by the following three rules:

$$\textsc{Comm} \quad \frac{l_1 : c \rightsquigarrow l_2 \in S \quad \mathcal{C}[\![c]\!]s_1 = s_2}{(l_1, s_1) \xrightarrow{c}_S (l_2, s_2)}$$

$$\textsc{Then} \quad \frac{l_1 : g \rightsquigarrow l_2 \ ; \ l_3 \in S \quad \mathcal{G}[\![g]\!]s_1}{(l_1, s_1) \xrightarrow{g}_S (l_2, s_1)}$$

$$\textsc{Else} \quad \frac{l_1 : g \rightsquigarrow l_2 \ ; \ l_3 \in S \quad \neg\mathcal{G}[\![g]\!]s_1}{(l_1, s_1) \xrightarrow{\overline{g}}_S (l_3, s_1)}$$

The action g (resp. $\overline{g}$) denotes the transition to the then-branch (resp. else-branch) of the corresponding conditional.

6 Availability aspects

Finite time properties are a common class of availability properties that ensure that users' requests are eventually answered. This type of liveness property must be ensured statically using verification techniques. They cannot be enforced dynamically by monitoring, weaving or code instrumentation [13]. Since only safety properties can be enforced by weaving, we consider *bounded time* properties which are availability and safety properties. For example, we may want to ensure that requests are answered before a fixed time limit. Many other timed properties can be specified as well. For instance, to guarantee a fair use of resources, we may want to limit the allocation frequency of resources by a service (*e.g.*, by adding waiting periods).

Availability aspects specify mostly maximal and minimal periods between events (*e.g.*, the allocation and release of a resource). They are written in a textual language and can be easily translated into timed automata.

6.1 Syntax

Our language is inspired by *stateful aspects* [14] (or trace-based aspects [15]) which take the history of execution into account. The syntax is described in Figure 8.

$$
\begin{array}{lcll}
A & ::= & \{a_i = E_i\} & \text{; mutually recursive equations} \\
\\
E & ::= & E_1 \,\square\, E_2 & \text{; choice} \\
 & | & ((F, G) \triangleright L); a_i & \text{; adds advice } L \text{ and proceeds} \\
 & & & \text{; with } a_i \text{ if the current event is} \\
 & & & \text{; matched by the pattern } F \text{ and} \\
 & & & \text{; the timed guard } G \text{ is true} \\
\\
F & ::= & Pat & \text{; basic event patterns} \\
 & | & F_1 \wedge F_2 \mid \neg F & \\
\\
G & ::= & \{\ldots, t \odot k, \ldots\} & \text{; timed guards} \\
 & & & \text{; } \odot \in \{\leq, <, >, \geq\} \\
L & ::= & \{I; \ldots; I\} & \text{; advice} \\
\\
I & ::= & \mathit{reset}\ (i, k) & \text{; programs the interrupt } i \text{ to be} \\
 & & & \text{; triggered in } k \text{ time units} \\
 & | & \mathit{cancel}\ (i) & \text{; cancels the interrupt } i \\
 & | & \mathit{start}\ (t) & \text{; initializes the timer } t \\
 & | & \mathit{wait}\ (t, k) & \text{; waits until } t = k \\
 & | & \mathit{nop} & \text{; empty instruction}
\end{array}
$$

with k an integer, i an interrupt and t a timer

Fig. 8. Syntax of availability aspects

An aspect is a collection of mutually recursive equations. An equation is of the form $a_i = (C \triangleright L); a_j$ and should be read as: the aspect waits for the event

C which triggers the execution of the sequence of instructions L and passes the control to equation a_j. In general, an equation may contain choices. For example, the aspect $(C \triangleright L); a \;\square\; (C' \triangleright L'); a'$ waits for the events C or C'; the first event occurring triggers the execution of the corresponding advice and equation (L and a or L' and a'). To ensure determinism, we suppose that choices are exclusive [1] .

A pattern F, close to AspectJ's pointcuts [16], is either a simple pattern (a term, possibly with wildcards `*`, matching commands), or a logical combination of patterns. For example, `R.alloc` matches only the allocation of the resource `R`, `*.alloc` matches all allocations and `R.*` all operations on the resource `R`. A guard G is a conjunction (represented by a set) of comparisons of timers to integer constants.

The list of instructions L denotes the advice to execute when the associated pattern matches the current instruction. Availability aspects use only 5 types of instructions:

- *reset*(i, k) programs an interrupt i to terminate the current request and to release all allocated resources after k seconds. We suppose that reset rolls back a service to a safe initial state (*e.g.,* using transactional techniques). Most resources (processor, memory, printer, *etc.*) can be adapted to support roll-back.
- *cancel*(i) cancels the interrupt i;
- *start*(t) initializes the timer t;
- *wait*(t, k) waits until t has the value k. If $t \geq k$ then the instruction does nothing ($\mathit{wait}(t, k) \equiv \mathit{nop}$);
- *nop* permits advance without performing any action.

All instructions are executed *after* the matched instruction (*i.e.*, they are after advice) except *wait*(t, k) which is performed *before* (*i.e.*, a before advice). We forbid programming and canceling the same interrupt (*e.g.*, *reset*(i, k); *cancel*(i)) within the same advice.

Availability aspects can only add guards or time-related instructions that do not modify the state of the service. Their semantic impact boils down to forbidding some execution traces: either they are aborted by a *reset* or their timing is modified by *wait*. Aspects can therefore be seen as timed properties and it is possible to reason on woven programs. Note that the instructions *reset* and *wait* are parameterized by time constants. Allowing variables is an easy extension but it would make verification (Section 8.2) undecidable. As illustrated by the examples in Section 6.2 and Section 10, fixed time limits

[1] Another option would be to choose the first choice (*i.e.*, C) when both choices match the same event

(or allocation frequencies) are common in availability policies. Constants are sufficient for that purpose.

To simplify notations, we omit the guard when it is *true* and list notation for a single instruction. For example, $(true, \texttt{M1.alloc}) \triangleright \{reset(i_1, 25)\}$ is written $\texttt{M1.alloc} \triangleright reset(i_1, 25)$.

6.2 Examples

We illustrate our language using several simple and common examples, namely controlling the duration of resource allocation, the frequency of resource allocations, the duration according to the frequency and, finally, enforcing a specific allocation ordering.

Controlling the duration of resource allocations We may want to weave the following two aspects to the service `S1` of Figure 3:

- A_1 that ensures that the resource `M1` is released within 25 seconds ;
- A_2 that ensures that the resource `M2` is released within 35 seconds.

These two aspects are specified as follows:

$$A_1 = \left\{ \begin{array}{l} a_1 = \texttt{M1.alloc} \triangleright reset(i_1, 25);\ a_2 \\ a_2 = \texttt{M1.free} \triangleright cancel(i_1);\ a_1 \end{array} \right\}$$

$$A_2 = \left\{ \begin{array}{l} a_1 = \texttt{M2.alloc} \triangleright reset(i_2, 35);\ a_2 \\ a_2 = \texttt{M2.free} \triangleright cancel(i_2);\ a_1 \end{array} \right\}$$

As soon as the event `M1.alloc` (resp. `M2.alloc`) is executed, a reset is programmed to be set off 25 seconds (resp. 35 seconds) later. If the event `M1.free` (resp. `M2.free`) occurs before, the interrupt is canceled.

Controlling the frequency of resource allocations Here, the goal is to prevent a service from monopolizing a resource by re-allocating it immediately. This may be required by resources constantly needed by several services.

Consider two services `X` and `Y` that need the resource `M` to answer a request. The service `X` tries to allocate `M1` as soon as it has released it whereas `Y` asks for it R seconds after it has started to process a new request. Better fairness can be guaranteed by making the service `X` wait at least 5 seconds between

each allocation of `M1`. The following aspect specifies such a property which will be woven on the service `S1`:

$$A_3 = \left\{ a_1 = \texttt{M1.alloc} \triangleright \{wait(t,5); start(t)\};\ a_1 \right\}$$

A wait of at least 5 seconds is imposed before a new event `M1.alloc` is performed ($wait(t,5)$). Afterward, the timer is reset and restarted. As for clocks in timed automata, we assume that all timers are initialized to 0 at the beginning of the program. Therefore, the aspect enforces that at least 5 seconds have passed between the beginning of the service and the first event `M1.alloc` .

Controlling the duration according to allocation frequency Instead of decreasing the frequency, another option is to adapt the allocation time depending on the frequency. For example, a policy might be to set the maximal allocation time to be 10 seconds except if the resource was already allocated by the same service less than 20 seconds before ($t < 20$). In that case, the maximal allocation time is only 5 seconds. The following aspect specifies that property:

$$\left\{ \begin{array}{l} a_1 = \texttt{M.alloc} \triangleright reset(i,10);\ a_2 \\ a_2 = \texttt{M.free} \triangleright \{cancel(i); start(t)\};\ a_1 \\ a_3 = (t < 20, \texttt{M.alloc}) \triangleright reset(i,5);\ a_2 \\ \quad \square\ (t \geq 20, \texttt{M.alloc}) \triangleright reset(i,10);\ a_2 \end{array} \right\}$$

Enforcing a resource allocation ordering Properties unrelated to time can also be specified using the same language. For instance, it is possible to enforce specific orders of resource allocation *e.g.*, to prevent deadlocks. The following aspect forbids the allocation of the resource `M1` if the service already possesses the resource `M2`. In this case, the service is terminated using $reset(i,0)$. This aspect is useful only for services which may allocate `M1` and `M2` in both orders. The aspect will select only executions allocating first `M1` then `M2`.

$$\left\{ \begin{array}{l} a_1 = \texttt{M2.alloc} \triangleright \{\};\ a_2 \\ a_2 = \texttt{M1.alloc} \triangleright reset(i,0);\ a_1 \\ \quad \square\ \texttt{M2.free} \triangleright \{\};\ a_1 \end{array} \right\}$$

Many other availability policies can be described in our language. For example, we could associate priorities to services and make them evolve according to

services' behavior. Different delays and frequencies could then be specified depending on the priority.

7 Weaving

Our approach implements weaving as a timed automata product. A service is represented by a timed automaton over-approximating its (timed) execution traces. The semantics of aspects is given as a timed automaton. Such an automaton recognizes the set of (timed) execution traces allowed by the aspect. The product of these two automata performs the intersection of their two sets of traces. That is, the product automaton recognizes the traces of the original service minus the traces forbidden by the aspect. In practice, it amounts to aborting some execution traces (using interrupts and resets) or to slowing down others (using waits).

We first describe how services are abstracted into timed automata. The abstraction consists in the control flow graph without any time constraints (*i.e.*, all the possible timing behaviors are included). Then, we give the semantics of aspects in terms of timed automata. The next step is to weave the aspect into the service. That step boils down to a classical product operation. The resulting automaton represents the service restricted in such a way that it respects the property specified by the aspect. This automaton might not be precise enough to verify availability properties. Section 8 will describe how to obtain a better automaton for verification purposes by taking timing information into account.

7.1 Abstraction of services

We use an abstraction over-approximating the execution traces (a standard control flow analysis) that does not take time information into account. This can be seen as the largest over-approximation as far as time is concerned. A service is represented by an automaton which can be seen as the control flow graph of the service. The timed execution of an instruction a is represented by three instants and transitions. The first instant/transition $\mathbb{I}(a)$ represents when the system knows the next instruction to be processed. The second instant/transition $\mathbb{B}(a)$ defines the time when the instruction really begins, and the third instant/transition $\mathbb{E}(a)$ the time when the instruction ends.

The state between $\mathbb{I}(a)$ and $\mathbb{B}(a)$ will be used to model possible *wait* advice. The state between $\mathbb{B}(a)$ and $\mathbb{E}(a)$ serves to model the duration of instructions. A *wait* advice also adds a timed constraint on the transition $\mathbb{B}(a)$.

The abstraction is described by the relation $\mathbf{next}_S(l_1, a, l_2)$ which denotes that S can go from the program point l_1 to l_2 by performing the action a. That relation is defined as follows:

$$\mathbf{next}_S(l_1, a, l_2) \textit{ iff } l_1 : a \rightsquigarrow l_2 \in S \ \vee \ l_1 : a \rightsquigarrow l_2 \ ; \ l \in S$$
$$\mathbf{next}_S(l_1, \overline{a}, l_2) \textit{ iff } l_1 : a \rightsquigarrow l \ ; \ l_2 \in S$$

The relation is clearly an over approximation of the control flow since values (and the evaluation of tests) are abstracted away.

The service $S = (\Sigma_S, (l_0, s_0), \mathcal{E}_S, \longrightarrow_S)$ is *abstracted* in the timed automaton $S^\sharp = (\Sigma_{S^\sharp}, l_0, \emptyset, \mathcal{E}_{S^\sharp}, \longrightarrow_{S^\sharp}, I_{S^\sharp})$ where

- $\Sigma_{S^\sharp}$, the set of abstract states, is composed of the set of program points and a set of intermediate states. Formally:

  $$\Sigma_{S^\sharp} = \{l, l_{a1}, l_{a2}, l' \ \mid \ \mathbf{next}_S(l, a, l')\}$$

- the initial abstract state is the initial label (program point) l_0;
- the set of clocks is empty;
- the set of actions is composed, for each action of S, of three actions (instants) $\mathbb{I}(a)$ (the initialisation of a), $\mathbb{B}(a)$ (the beginning of a) and $\mathbb{E}(a)$ (the end of a):

  $$\mathcal{E}_{S^\sharp} = \{\mathbb{I}(a), \mathbb{B}(a), \mathbb{E}(a) \ \mid \ a \in \mathcal{E}_S\}$$

  Splitting the action in three instants is used to represent the timed execution of an instruction;
- the transition relation $\longrightarrow_{S^\sharp}$ is defined as follows:

  $$\begin{aligned}&(l, ., \mathbb{I}(a), ., l_{a1}) \in \longrightarrow_{S^\sharp} \wedge (l_{a1}, ., \mathbb{B}(a), ., l_{a2}) \in \longrightarrow_{S^\sharp} \\ &\wedge \ (l_{a2}, ., \mathbb{E}(a), ., l') \in \longrightarrow_{S^\sharp} \\ &\textit{iff} \quad \mathbf{next}_S(l, a, l')\end{aligned}$$

  Each action a from one state to another is represented using two intermediate states l_{a1} and l_{a2}, and three transitions corresponding to the three instants $\mathbb{I}(a)$, $\mathbb{B}(a)$ and $\mathbb{E}(a)$ without any timing constraint for now.
- the function $I_{S^\sharp}$ does not add any timing constraint, that is:

  $$\forall l \in \Sigma_{S^\sharp}. \ I_{S^\sharp}(l) = \emptyset$$

The absence of any timing constraint implies that the automaton models all possible execution times for each action. Figure 9 illustrates the abstraction of service `S1` into a timed automaton.

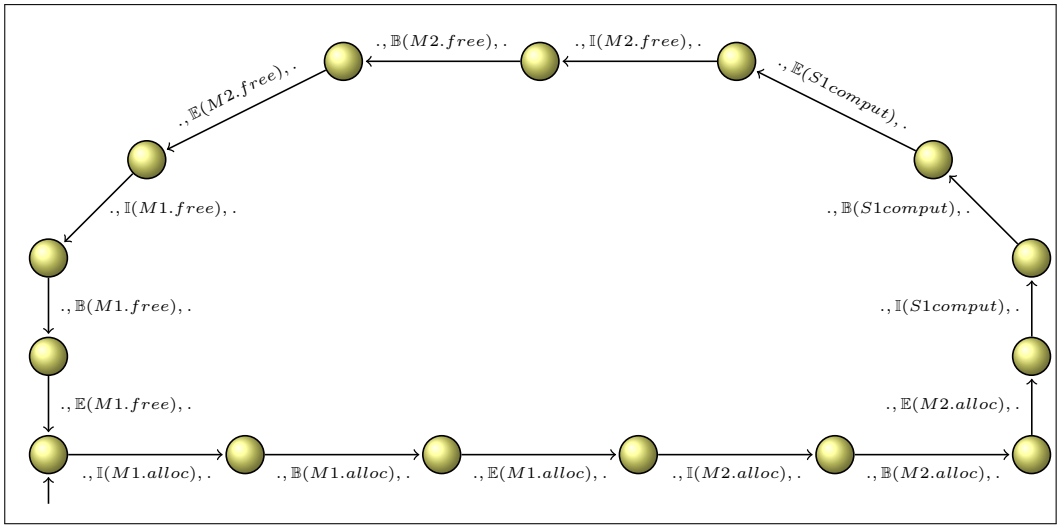

Fig. 9. Abstraction of service `S1`

The abstraction is safe since the automaton accepts all execution traces of the source program. Formally:

PROPERTY **1** [Safety] *A service* $S = (\Sigma_S, (l_0, s_0), \mathcal{E}_S, \longrightarrow_S)$ *and its associated abstraction* $S^\sharp = (\Sigma_{S^\sharp}, l_0, \mathcal{E}_{S^\sharp}, \longrightarrow_{S^\sharp}, I_{S^\sharp})$ *are such that for all labels* l_1 *and* l_2*, states* s_1 *and* s_2*, and action* a*:*

$$(l_1, s_1) \xrightarrow{a}_S (l_2, s_2) \Rightarrow \exists l_{a1}, l_{a2}.\ l_1 \xrightarrow{.,\mathbb{I}(a),.}_{S^\sharp} l_{a1} \wedge l_{a1} \xrightarrow{.,\mathbb{B}(a),.}_{S^\sharp} l_{a2} \wedge l_{a2} \xrightarrow{.,\mathbb{E}(a),.}_{S^\sharp} l_2$$

7.2 Aspect semantics

The semantics of aspects is given in terms of timed automata. An aspect specifies a timed property and the timed traces recognized by the corresponding semantic automaton are the timed traces allowed by the aspect. Intuitively, the different basic advice instructions can be described as follows:

- $reset(i, k)$ starts a timer i to abort the current request after k time units. The timer i is reset on the $\mathbb{E}()$ transition to the instruction matched by the corresponding pointcut. After the $\mathbb{E}()$ transition, the interrupt environment, which records the set of active *reset*s, associated the timer i to k.
- $cancel(i)$ removes i from the interrupt environment (*i.e.*, associates $\perp$ to i);
- $wait(t, k)$ adds the guard $t \geq k$ at the beginning of the action;
- $start(t)$ resets the timer t on the $\mathbb{E}()$ transition to the instruction matched by the corresponding pointcut.

The semantics of aspects is given by automata of the form:

$$A = (N_a, l_{a0}, H_a, \mathcal{E}_a, \longrightarrow_a, I_a) \qquad \text{where}$$

- the set of states N_a is made of a sink state RESET and pairs (q, e) where q denotes the state (*i.e.*, the current equation) of the aspect and e the current interrupt environment;
- $l_{a0} = (a_0, \{\})$ is the initial state;
- H_a is the set of clocks (interrupts and timers) used in the aspect;
- $\mathcal{E}_a$ contains the same actions as the service;
- I_a associates each state (q, e) to an invariant enforcing that no valid interrupt (*i.e.*, defined in e) occurs. This function is defined as follows:

$$I_a(q, e) = \{i \leq e(i) \mid \forall i.e(i) \neq \perp\}$$

In the remainder of this section, we use the special transition $(q, e) \xrightarrow{else}_a (q, e)$ which denotes that if no other transitions from (q, e) applies then the aspect remains in the same state. This notation is syntactic sugar which can be translated into a collection of transitions from (q, e) to (q, e) (the complementary of outgoing transitions).

The relation $\longrightarrow_a$ is defined on the syntax of the aspect as follows:

$$[a_0 = E_0] = (a_0, \{\}) \xrightarrow{else}_a (a_0, \{\}) \quad \cup \quad [E_0]^{(a_0,\{\})}$$

The automaton corresponding to E_0 (the initial equation) has the initial state $(a_0, \{\})$. No interrupt is active and, as for all states, there is an $else$ transition.

$$[E_1 \Box E_2]^{(q,e)} = [E_1]^{(q,e)} \cup [E_2]^{(q,e)}$$

The transitions corresponding to an exclusive choice are the union of the transitions for both choices.

$$\begin{aligned}
&[(F, G) \triangleright L; a_i]^{(q,e)} \\
&= [(F, G) \triangleright L]^{(q,e)}_{(a_i,e')} \quad \cup \quad [E_i]^{(a_i,e')} \qquad\qquad (\{a_i = E_i\} \in A) \\
&\quad \cup \quad (a_i, e') \xrightarrow{else}_a (a_i, e') \quad \cup \quad interrupt(a_i, e')
\end{aligned}$$

A rule $(F, G) \triangleright L$ involves the computation of a new interrupt environment (see the next translation rule) and new transitions to a new state. The automaton corresponding to the continuation of the aspect starts from this new state. As any state, the $else$ transition and the interrupt transitions (contained in the

current environment) are generated.

$$
\begin{aligned}
&[(F, G) \triangleright L]^{(q_1,e_1)}_{(q_2,e_2)} \\
&= \{\ (q_1, e_1) \xrightarrow{G \cup \bigwedge(e1), \mathbb{I}(a),.}_a (q_{a1}, e_1) \quad \cup \quad (q_{a1}, e_1) \xrightarrow{g_i \cup \bigwedge(e1), \mathbb{B}(a),.}_a (q_{a2}, e_1) \\
&\quad \cup (q_{a2}, e_1) \xrightarrow{\bigwedge(e1), \mathbb{E}(a), r}_a (q_2, e_2) \\
&\quad \cup \quad \mathit{interrupt}(q_{a1}, e_1) \quad \cup \quad \mathit{interrupt}(q_{a2}, e_1) \\
&\quad \mid \ \mathit{match}(a, F) \ \wedge \ \mathit{ins}(e_1, L) = (g_i, r, e_2)
\end{aligned}
$$

For each action a matched by F, three transitions ($\mathbb{I}(a)$, $\mathbb{B}(a)$ and $\mathbb{E}(a)$) are added using two new intermediate state (q_{a1}, e_1) and (q_{a2}, e_1). Transitions modeling interrupts are added to these states. The function *ins* analyzes the advice L to compute the guards and resets of timers as well as the new interrupt environment e_2. The guard g_i represents the constraints for the *wait* in the advice and the set r represents the timers reset by *reset* and *start* instructions in the advice.

The intermediate functions used in the translation are defined as follows:

- The function *interrupt* takes a state (q, e) and returns the set of transitions modeling the interrupts that may arise in this state.

$$\mathit{interrupt}(q, e) = \{(q, e) \xrightarrow{i \geq e(i),...,}_a \ \textsc{Reset} \mid e(i) \neq \bot\}$$

  There is a transition to Reset each time an interrupt i reaches its trigger value recorded in the environment e.
- The function $\mathit{match}(a, F)$ returns true if F matches a.
- The function *ins* takes an interrupt environment, an advice and returns the guard, the reset set and the new interrupt environment taking into account the *wait*, *reset*, *start* and *cancel* instructions of the advice.

$$
\begin{aligned}
\mathit{ins}(e, L) = (&\{t \geq k \ \mid \ \mathit{wait}(t, k) \in L\}, \\
&\{z \mid \mathit{reset}(z, k) \in L \vee \mathit{start}(z) \in L\}, \\
&e')
\end{aligned}
$$

$$
\text{with} \quad \begin{cases}
e'(i) = \bot & \text{if } \ \mathit{cancel}(i) \in L \\
e'(i) = k & \text{if } \ \mathit{reset}(i, k) \in L \\
e'(i) = e(i) & \text{otherwise}
\end{cases}
$$

- The function $\bigwedge$ takes an environment and returns the guard corresponding to the case where no interrupt occurs: $\bigwedge(e) = \{i < e(i) \ \mid \ e(i) \neq \bot\}$

The translation proceeds by unfolding the recursive equations of the aspect. The process terminates since there are a finite number of definitions ($a_i = \ldots$) and interrupt environments.

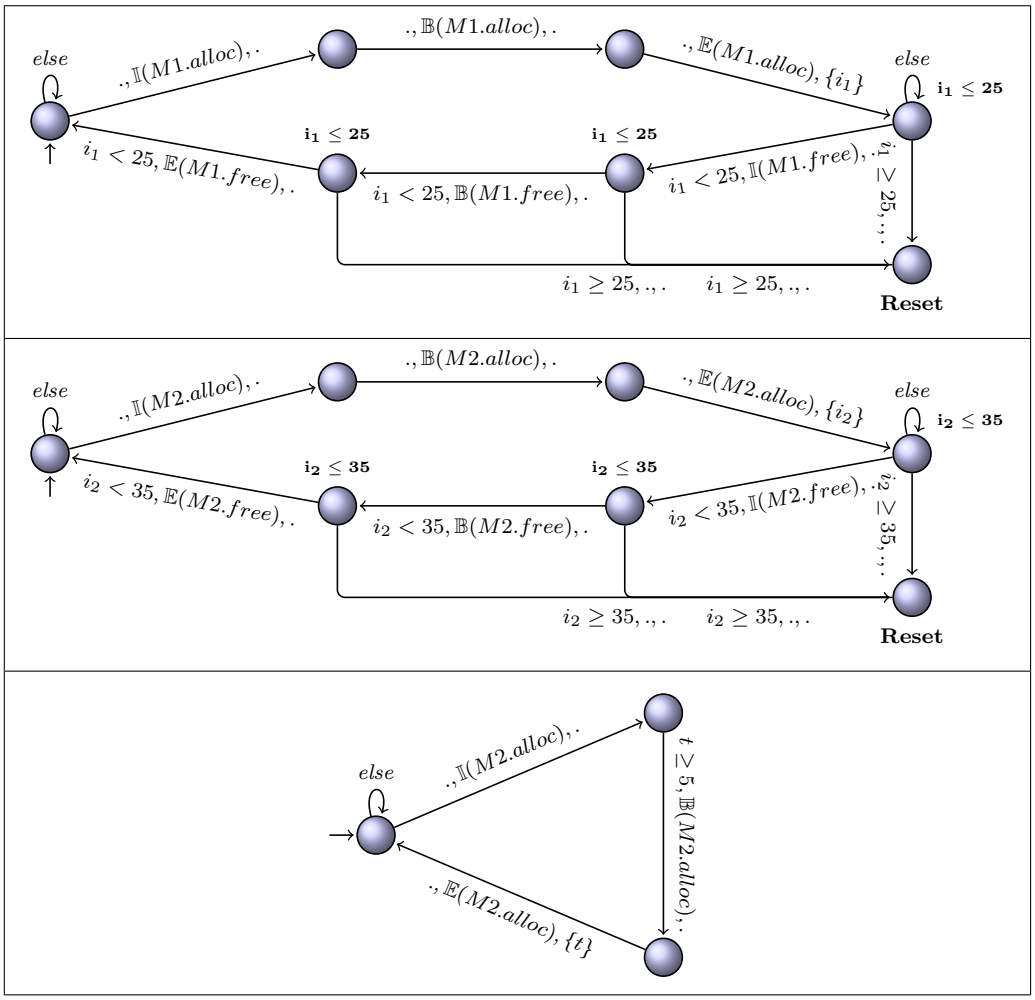

Fig. 10. Timed automata of A_1 (above), A_2 (middle) and A_3 (below)

Figure 10 shows the semantic automata for the previously defined aspects A_1, A_2 and A_3. In the aspect A_1, the clock i is reset at the initialization of the interrupt. Then, for all states until the resource is released, the outgoing transitions have the guard $i < 25$, the state invariant has the condition $i < 25$ and a transition with guard $i \geq 25$ to the state Reset is added. The sink state Reset will be interpreted during the concretization as a collection of transitions releasing all resources followed by a transition returning to the beginning of the request loop.

Intuitively, the weaving of the first two aspects will amount to starting a timer when the service takes the resource and to resetting the service when the timer reaches its time limit (*i.e.*, 25 or 35 seconds). For the aspect A_3, weaving will ensure that there are at least 5 seconds between two `M1.alloc` events. This behavior is simply described by the guard $t \geq 5$ on transition $\mathbb{I}(M1.alloc)$ and

by resetting the timer t after each `M1.alloc` *i.e.*, at the transition $\mathbb{E}(M1.alloc)$.

7.3 Weaving an aspect to a service

In aspect oriented programming terminology, weaving is the step which inserts advice within the program. Weaving *per se* is just the product (as described in Section 3.3) of the automata representing the service and the aspect. This practical and theoretical simplicity of weaving is an important benefit of our framework. The aspect automaton specifies a set of allowed timed traces using timers, guards and invariants. The automata product performs the intersection of the execution traces of the service and aspect. The semantic impact of weaving is therefore to restrict the service's behavior to the timed traces allowed by the aspect. In implementation terms, it amounts to inserting the time annotations of the aspect within the service to shorten or lengthen some timed executions.

Figure 11 shows the product of the abstraction of service `S1` with the aspects A_1, A_2 and A_3. In the product automaton, two interrupts are programmed after `M1.alloc` and after `M2.alloc`, and one timer is started after `M1.alloc`. If `M1.free` (resp. `M2.free`) is not executed before 25 seconds (resp. 35 seconds), the automaton goes to state RESET. `M1.alloc` is also constrained by $t \geq 5$ which enforces to wait at least 5 seconds between two calls to `M1.alloc`.

In comparison with Figure 9, guards, transitions to RESET and state invariants have been added to model interrupts and timers.

Compared to a standard weaving a la AspectJ, the final result is similar: new code (*i.e.*, advice) is added at various join points. The respective approaches are however quite different. In AspectJ, design and reasoning are mainly syntactic processes. Aspects specify sets of join points and code to insert at these points. The programmer usually reasons on the semantics of the program by (mentally) visualizing the expected source code of the woven code. In our domain-specific language, where advice is restricted, aspects can be seen as a (timed) property on execution traces. An aspect specifies a set of allowed traces which can be enforced to the base program using automata product and a concretization into source code.

8 Optimization, verification and concretization

The product (woven) automaton can be

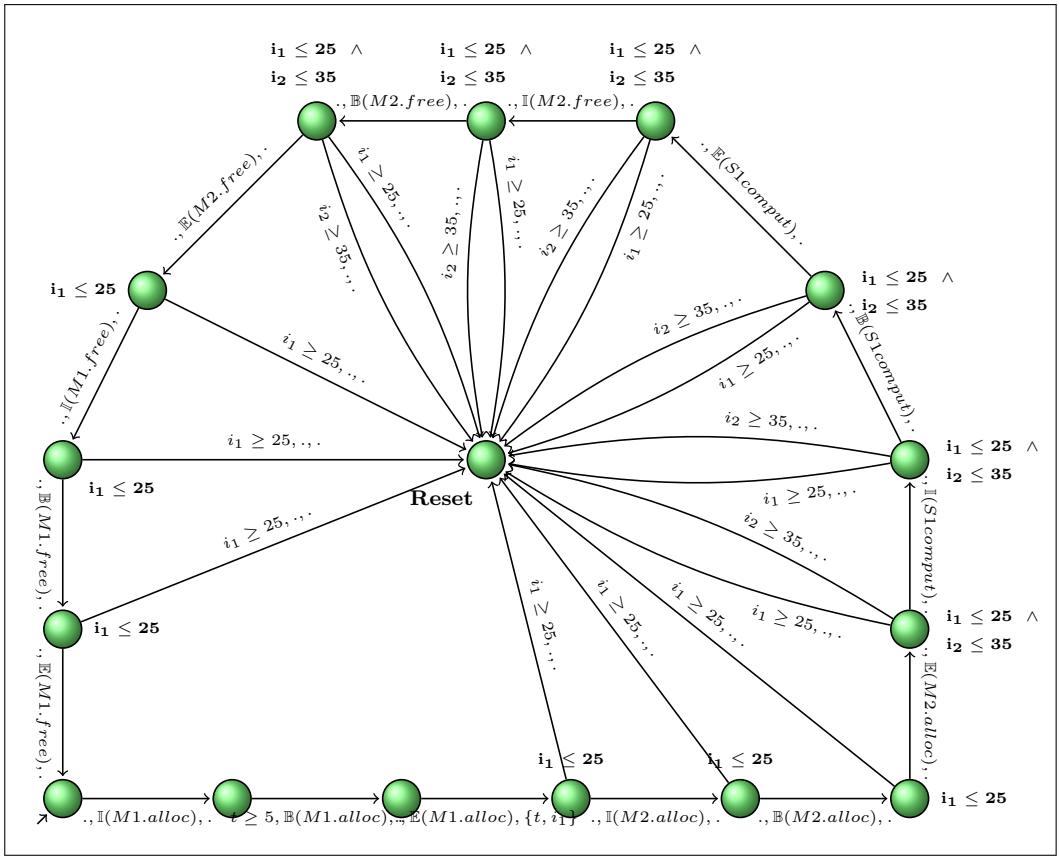

Fig. 11. Product of service `S1` with aspects A_1, A_2 and A_3

- optimized by taking into account (worst-case and best case) execution times of instructions, and by removing all useless delays;
- used to model-check general availability properties (*e.g.*, absence of deadlock, boundedness of the request loop, *etc.*);
- translated back into a source program.

We briefly present these three steps in turn.

8.1 Optimizations

We describe here how to optimize the woven automaton by taking into account execution time of instruction. We assume a cost function f_{cost} returning for each instruction of the service a time interval $[\text{BCET}(I), \text{WCET}(I)]$ where $\text{BCET}(I)$ (resp. $\text{WCET}(I)$) is a best-case (resp. worst-case) execution time of I. Note that it is always possible to build such a function since the trivial approximation $f_{cost}(I) = [0, +\infty]$ is always safe (if not very useful). Such intervals can be seen as a new constraint removing all execution traces where I takes less (resp. more) than BCET (I) (resp. WCET (I)). Again, these constraints are taken into account by a product operation. A precise cost function (*e.g.*,

see [17,18]) permits the removal of spurious tests or useless timers from the woven automaton. For instance, if f_{cost} directly implies that a service releases its resource before the time limit required by an aspect, no instrumentation will be needed to enforce this requirement.

In the following, we suppose that we have such a cost function and that it returns the following results for the instructions of service `S1`:

$$\begin{aligned} f_{cost}(\texttt{S1Comput}) &= [2, 10] \\ f_{cost}(\texttt{M1.alloc}()) = f_{cost}(\texttt{M2.alloc}()) &= [0, +\infty] \\ f_{cost}(\texttt{M1.free}()) = f_{cost}(\texttt{M2.free}()) &= [0, 0] \end{aligned}$$

The function f_{cost} yields an unbounded time interval for allocations since these instructions depend on the state of the resource and are blocking. The time information is taken into account by performing a product with the *cost automaton* $C = (N_c, c_0, \{k\}, \mathcal{E}_{S^\sharp}, \longrightarrow_c, I_c)$ where:

- for any action a such that $f_{cost}(a) = [\text{BCET}(a), \text{WCET}(a)]$ we have

  $$c_0 \xrightarrow{.,\mathbb{I}(a),.}_c q_{a1}, \quad q_{a1} \xrightarrow{.,\mathbb{B}(a),\{k\}}_c q_{a2} \quad \text{and} \quad q_{a2} \xrightarrow{k \geq \text{BCET}(a),\mathbb{E}(a),.}_c c_0$$

  with q_{a1} and q_{a2} fresh states

- the state invariant specifies that control can remain in this state not longer than $\text{WCET}(a)$; that is:

  $$I_c(q) = \begin{cases} \{k \leq \text{WCET}(a)\} & \text{if } \exists\, q \xrightarrow{k \geq \text{BCET}(a),\mathbb{E}(a),.}_c c_0 \ \in \longrightarrow_c \\ \emptyset & \text{otherwise} \end{cases}$$

The timer k is reset at the beginning of a. The control remains in the intermediate state at least until $k \geq \text{BCET}(a)$ and at most until $k = \text{WCET}(a)$.

Figure 12 shows this automaton for service `S1`. We have not represented transitions corresponding to *alloc* because we do not have useful time information about this instruction.

Another issue to take into account is that sequencing (*i.e.*, the ; operator) takes no time. In our framework, this fact can be taken into account by a product with a two-state timed automaton, the *sequencing automaton*, $E = (\{e_0, e_1\}, e_0, \{seq\}, \mathcal{E}_{S^\sharp}, \longrightarrow_e, I_e)$ where:

- each beginning of action goes to state e_1 and each end of action goes to e_0 resetting the dedicated timer *seq*. Intuitively, the state e_0 represents the sequencing between actions (which takes no time) and the state e_1 represents

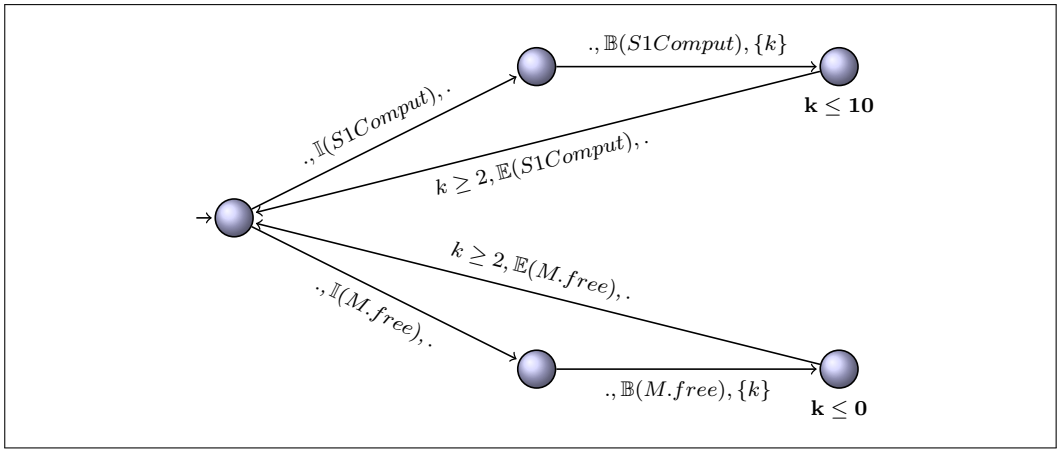

Fig. 12. Automaton constraining execution times of instructions of service `S1`

an action which may take time.

$$\longrightarrow_e = \left\{ \begin{array}{l} e_0 \xrightarrow{.,\mathbb{I}(a),.}_e e_1 \\ e_1 \xrightarrow{.,\mathbb{B}(a),.}_e e_2 \\ e_2 \xrightarrow{.,\mathbb{E}(a),\{seq\}}_e e_0 \end{array} \middle| \; \mathbb{B}(a) \in \mathcal{E}_{S^\sharp} \wedge \mathbb{E}(a) \in \mathcal{E}_{S^\sharp} \right\}$$

- the invariant of state e_0 ensures that no time can be spent in this state. No constraint is placed on state e_1.

$$I_e(e_0) = \{seq \leq 0\}, \;\; I_e(e_1) = \emptyset \;\; \text{and} \;\; I_e(e_2) = \emptyset$$

Figure 13 shows this automaton for service `S1`. To simplify the automaton, we use the special symbol X which denotes all instructions of the service.

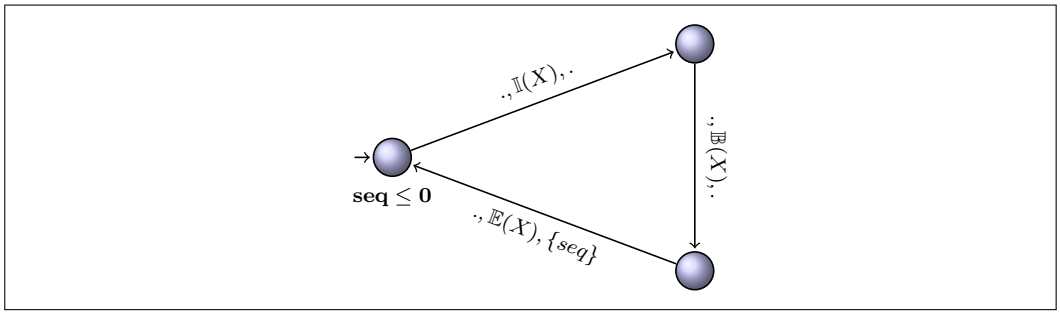

Fig. 13. Automaton for sequencing service `S1`

The last optimization step consists in taking $\mathbb{B}()$ transitions as early as possible. This is a bit more difficult because this transition can be constrained by $t \geq k$ guards corresponding to *wait* advices. Each instruction of the woven program described by the three transitions

$$q_0 \xrightarrow{g_0,\mathbb{I}(a),r_0} q_1 \quad q_1 \xrightarrow{g_1,\mathbb{B}(a),r_1} q_2 \quad q_2 \xrightarrow{g_2,\mathbb{E}(a),r_2} q_3$$

is transformed using the two following steps:

- the state q_1 is made urgent by resetting the clock u on the transition from q_0 to q_1 and by having the state invariant $I(q_1) = \{u \leq 0\}$.
- a loop is added for each guard $(t \geq k) \in g_1$

  $$q \xrightarrow{(t<k),.,.} q_w \quad q_w \xrightarrow{(t\geq k),.,u} q$$

  By definition of abstraction of services, aspect semantics and product of timed automata, these $(t \geq k)$ guards represent *wait* constraints. The state invariant of the state q_w is defined by $I(q_w) = \{t \leq k\} \cup I(q)$. The transitions from the state q_1 into RESET are also duplicated on state q_w.

For example, when in state q_1 with a guard $t \geq k$, then either the guard is satisfied and the action a is performed immediately, or the automaton performs an immediate transition into the new state q_w where time passes until $t \geq k$. When $t = k$, the transition goes back into state q_1 where a is performed immediately. This case generalizes to any number of wait constraints.

The timed automaton obtained after the product with the cost and sequencing automata and the transformation to take earliest the $\mathbb{B}$ transition is more precise. These optimizations have integrated time information and have removed many impossible timed traces The resulting automaton can be analyzed to remove useless guards, timers and invariants as well as unreachable states. This process optimizes the overhead introduced by the aspect. It is easily carried out by tools such as UPPAAL.

Figure 14 shows the service `S1` of Figure 11 after computing the product with the sequencing and cost automata corresponding to f_{cost}, and simplification.

Aspect A_2 prevents the service from retaining the resource `M2` more than 35 seconds. The weaving of A_2 has no impact on the code since the automaton makes it clear that `S1Comput` (*i.e.*, the use of `M2`) lasts at most 10 seconds. This information, initially given by f_{cost} and integrated by product in the service automaton, permits suppression of the useless interrupt i_2 and the related transitions.

8.2 Verification

The previous product automaton is a formal representation of the woven service. We may now want to verify that woven services satisfy general availability properties that are not directly specified by aspects. Actually, aspects are best seen as collections of timed properties (or availability policies) which are supposed to ensure high-level availability properties. These properties can be

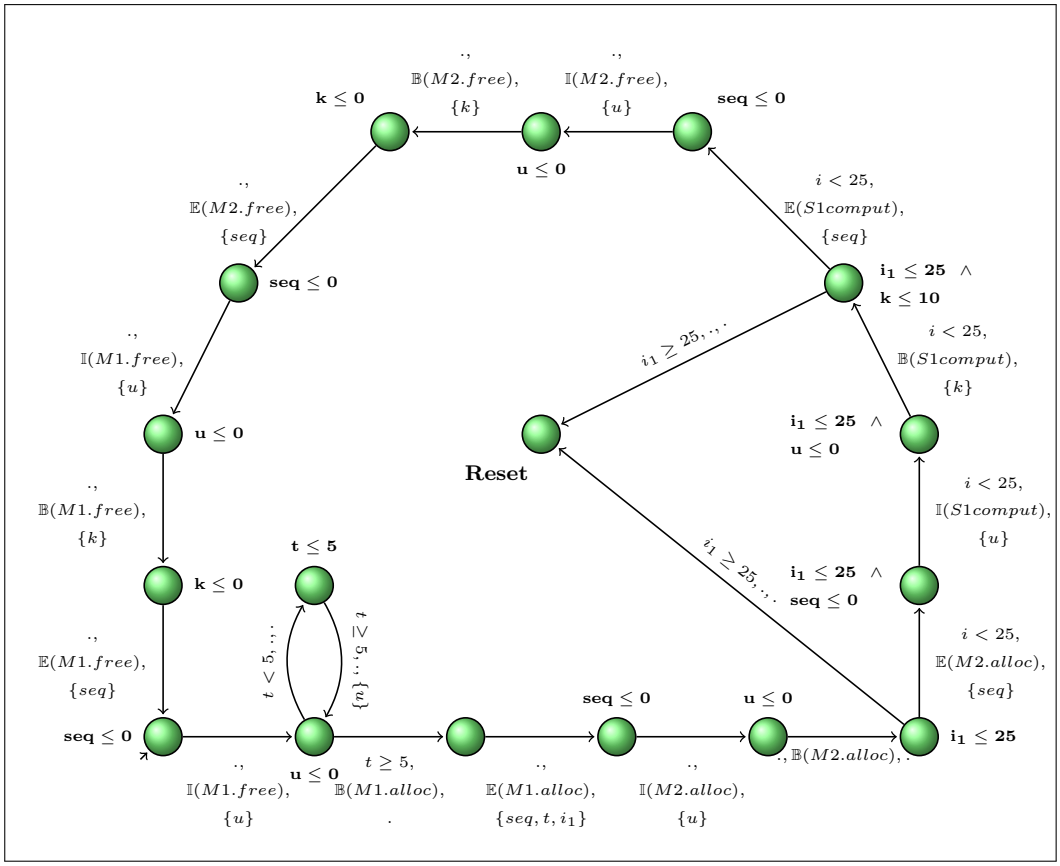

Fig. 14. Timed automaton of `S1` after weaving and optimization

verified by *model-checking* on the woven automaton. This verification step allows also checking that aspects are not contradictory. For example, an aspect adding waiting periods (*e.g.*, to lower an allocation frequency) may conflict with another aspect limiting the duration of another resource allocation. It is also possible to verify global properties (*e.g.*, absence of deadlocks) on the complete system composed of the woven services and shared resources.

We have used UPPAAL to represent services and to verify properties expressed as LTL formulas. We have woven the aspects A_1, A_2 and A_3 on service `S1`. We have applied similar aspects on service `S2`; namely two aspects enforcing the releasing of resource `M1` before 10 seconds and resource `M2` before 15 seconds. We have verified that the woven system respects the following properties:

- *the system is well timed and has no deadlock*. Note that deadlocks are prevented by the aspect resetting `S2` after 35 seconds;
- *the service* `S1` *treats a request in less than* 35 *seconds*. This property can be verified using a new timer `avail` reset at the beginning of the request loop and by checking `avail`≤ 35 for all states. This property has been ensured by weaving. Indeed, `S1` cannot wait more than 25 seconds to get its resources (10 seconds for resource `M1` and 15 seconds for resource `M2`) the woven service `S2` must release resource `M2` at most after 35 seconds. Since `S1Comput` takes

at most 10 seconds, `S1` will terminate before 35 seconds. This also means that service `S1` will always get access to the needed resources and, more generally, that no denial of service `S1` can arise anymore.

The verification of these properties is very fast (less that 1 second). Since UPPAAL has been used to analyze complex protocols, we expect that it could verify availability properties of much larger systems.

8.3 Concretization

The concretization of a standard automaton into our source code is very simple [19]. The concretization of timed automata requires the introduction of timed instructions (initialization of timers, checking time invariants, timed guards).

In order to take into account the timing facet introduced in the automaton during weaving, we extend our source language with timed guards and commands.

Guards are extended with timer comparisons:

$$g ::= t \odot k \;\mid\; \ldots \text{ with } \odot \in \{<, >, \leq, \ldots\}$$

The following commands are added:

$$c ::= \mathit{start}(t) \;\mid\; \mathit{wait}(t,k) \;\mid\; \mathit{reset}(i,k) \;\mid\; \mathit{cancel}(i) \;\mid\; \ldots$$

where t and i denote identifiers for a timer and an interrupt, respectively, and k denotes an integer. These commands are the source code equivalent of the advice instructions. The $\mathit{start}(t)$ command sets and starts a timer t which could be compared to integer constants in guards. Timers are also used to slow down an execution using the command $\mathit{wait}(t,k)$ that waits while $t < k$. The $\mathit{reset}(i,k)$ command programs an interrupt i to arise after k seconds. The $\mathit{cancel}(i)$ instruction cancels the interrupt i. The commands are the equivalent in source code of the advice instructions.

We sketch how a timed automaton is translated into that extended language. First, the time information introduced by the cost and sequencing automata is removed since it does not describe program instructions but merely non-functional properties. Concretization uses the following rules:

- pairs of transitions of the form

$$(q_1 \xrightarrow{.,\mathbb{I}(a),r} q_2, \quad q_2 \xrightarrow{.,\mathbb{B}(c),.} q_3, \quad q_3 \xrightarrow{.,\mathbb{E}(c),.} q_4)$$

correspond to a command c and are translated into the instruction $l_{q1} : c \rightsquigarrow l_{q4}$;

- pairs of transitions of the form

$$\begin{array}{c}(\; q_1 \xrightarrow{.,\mathbb{I}(g),.} q_2, \quad q_2 \xrightarrow{.,\mathbb{B}(g),.} q_3, \quad q_3 \xrightarrow{.,\mathbb{E}(g),.} q_4 \\ \quad q_1 \xrightarrow{.,\mathbb{I}(\overline{g}),.} q'_2, \quad q'_2 \xrightarrow{.,\mathbb{B}(\overline{g}),.} q'_3 \quad q'_3 \xrightarrow{.,\mathbb{E}(\overline{g}),.} q'_4)\end{array}$$

  correspond to a guard g and are translated into the instruction $l_{q1} : g \rightsquigarrow l_{q4} \; ; \; l_{q4'}$;

- pairs of transitions of the form

$$(q_1 \xrightarrow{g \wedge G,\mathbb{I}(a),.} q_2, \; q_1 \xrightarrow{\neg g \wedge G,\mathbb{I}(a),.} q_3)$$

  correspond to a guard g added by an aspect and are translated into the instruction $l_{q1} : g \rightsquigarrow l_{q2} \; ; \; l_{q3}$. Concretization proceeds with the transitions

$$(q'_1 \xrightarrow{G,\mathbb{I}(a),.} q'_2, \; q'_1 \xrightarrow{G,\mathbb{I}(a),.} q'_3)$$

- a loop $q \xrightarrow{t<k,.,.} q' \xrightarrow{t \geq k,.,seq} q$ involves the insertion of the command $wait(t, k)$ *before* the corresponding program point (*i.e.*, l_q);
- the reset of a timer t in a transition $q \xrightarrow{g,\mathbb{E}(a),\{t\}} q'$ is translated by the insertion of a command $start(t)$ after the program point corresponding to q' (*i.e.*, $l_{q'}$);
- an interrupt involves inserting the command $reset(i, k)$ at the initialization of i (*i.e.*, i is within a reset) and the command $cancel(i)$ at the program point corresponding to the first state where there is no invariant $i \leq k$ anymore.

Figure 15 shows the source code of service `S1` obtained after the concretization of the automaton of Figure 14. After the command `M1.alloc()`, a new interrupt `i` is set to arise after 25 seconds. When the service takes less than 25 seconds to complete, the resource `M1` is released (`M1.free()`) and the interrupt is canceled (*cancel* (i)).

9 Implementation issues

We have previously implemented related techniques [19,20] based on similar steps (abstraction, weaving, optimization, concretization) in a simpler, untimed, setting. The source language was a simple Pascal-like imperative language and aspects were safety properties expressed as finite state automata. The extension to timed properties has not been implemented and the experiments related in this article were conducted manually. In this section, we

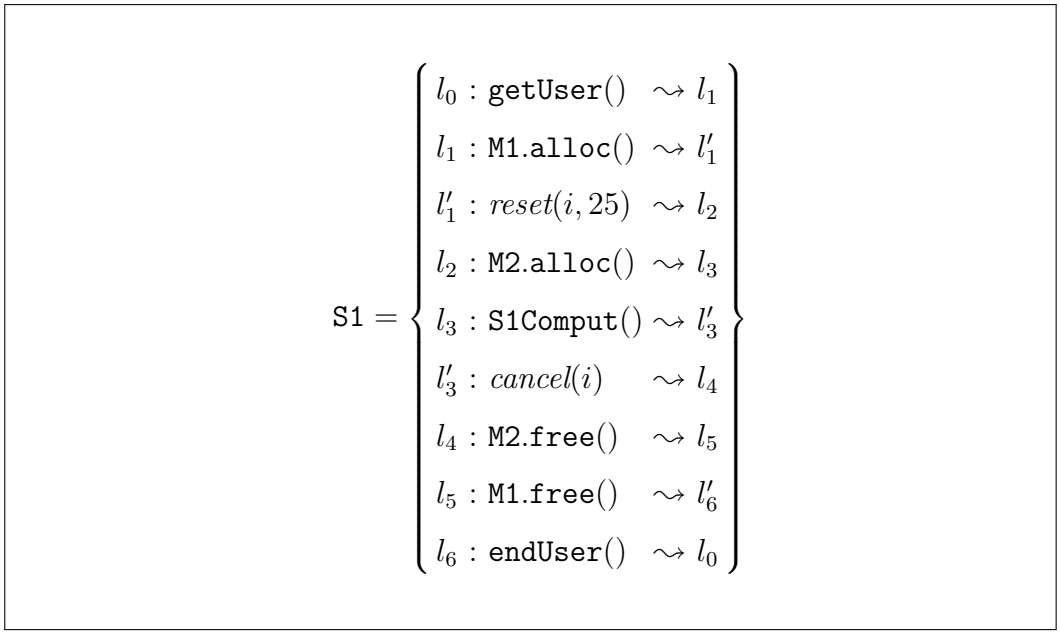

Fig. 15. Code of service `S1` after weaving

review implementation issues raised by availability aspects. We focus on the core of the technique, that is to say the abstraction, weaving, optimization, verification and concretization steps.

Abstraction Abstraction relies on a control flow analysis (CFA) to produce a safe control-flow automaton. This step makes the rest of the approach independent of the source programming language. In particular, our approach is not limited, nor specialized, to the simple imperative language used in the article.

There are many CFA variants and it is always possible to trade precision for efficiency. For first order languages, determining the control flow graph is considered almost trivial and most research on CFAs consider higher-order languages whose complexity goes from polynomial to exponential time [21]. It is easy to design linear-time analyses producing a safe control-flow automaton for a standard imperative language. However, modeling procedure calls and returns by a timed automaton is a crude approximation of procedural programs. In [20] we extend finite automata with return stacks to represent inter-procedural control flow more precisely. A similar extension of timed automata, as well as the corresponding weaving, optimization and concretization steps, could certainly be designed. However, it is unlikely that the verification process could be extended to tackle these new automata.

Note that only instructions related to resource management need to appear in the control flow graph. Parts of the program that do not include such instructions may often be summarized by a single event. Therefore, the automaton

produced by the abstraction is typically much smaller than the base program itself.

Weaving Weaving *per se* is a timed automata product. It is a simple operation but it may cause a blow up of the number of nodes of the resulting automaton. Even if the automata representing the aspects and the program are small, the multiplication of nodes may result in duplicating large chunks of the base program during concretization. We have previously proposed a method preventing all duplications in the context of finite state automata [20]. We describe the main ideas of this technique and how it can be adapted to timed automata on small examples.

Figure 16 presents a small example of weaving a safety property using finite state automata and product. It comprises:

(a) a base program automaton whose traces belong to the language $(aba^*b)^*$;
(b) an aspect enforcing that each event a is followed immediately by the event b. Otherwise the program must be reset;
(c) the standard automata product where the program is reset if the loop a^* is taken more than once.

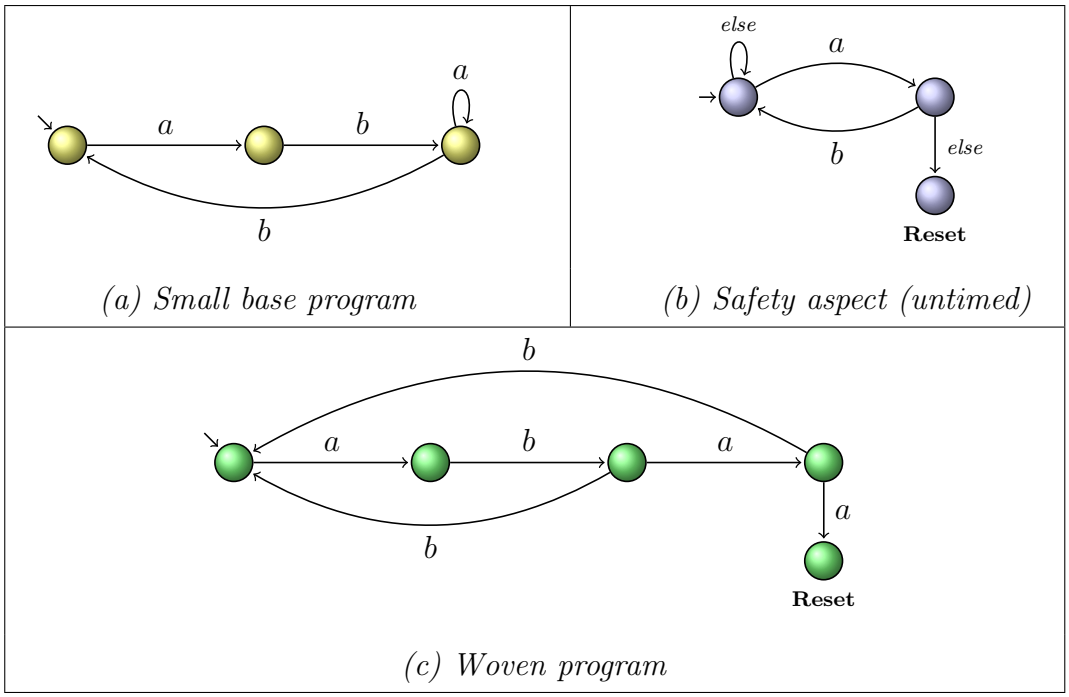

Fig. 16. Weaving as a standard automata product

Apart from the special **Reset** node, the woven program has one more state (*i.e.*, a new control point) than the base program. In general, the woven automaton may have $n \times m$ states where n and m are the number of states of

the base program and aspect automata respectively.

To prevent duplication, we define a so-called *instrumented product* where the state of the aspect automata is encoded and manipulated as an integer variable. Figure 17 (*a*) presents the direct instrumented product of the previous example. The automaton keeps the same number of states (except for **Reset**). Instead, the automaton is equipped with additional structures (a state variable, guards and assignments) to mimic the aspect automaton. The variable s represents the state of the aspect (initially 1). Each transition of the base program tests s and makes it evolve as if the aspect was executed in parallel. For instance, the loop-state can execute a if the aspect is in state 1 (in which case it goes to state 2), or performs a reset if the aspect is in state 2. That automaton is easily encoded in UPPAAL which allows guards and assignments on bounded integer variables.

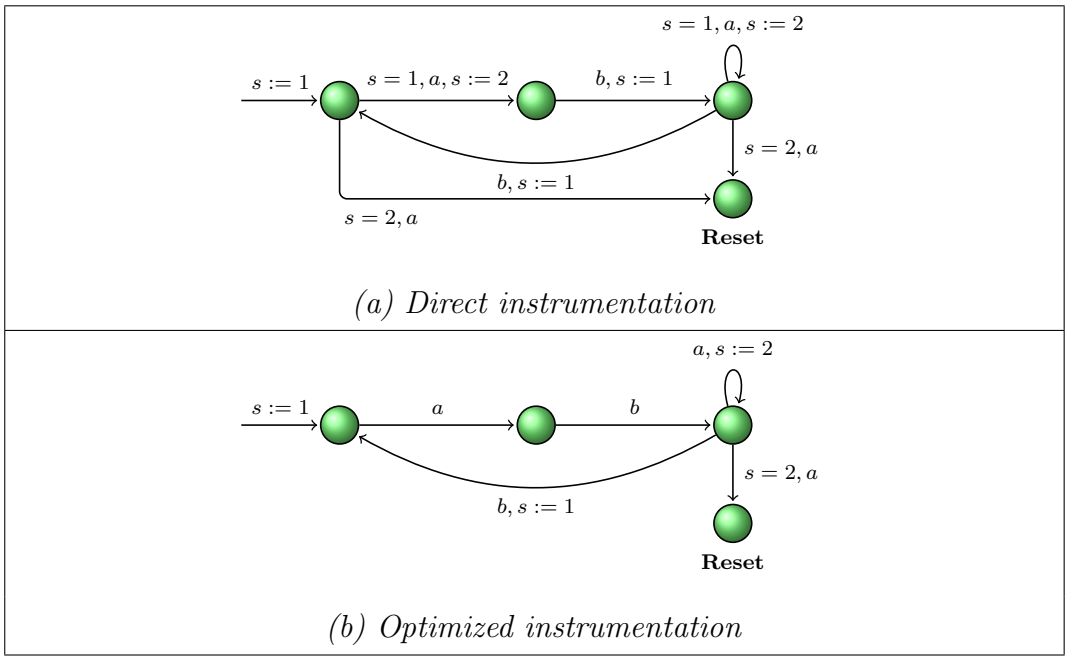

Fig. 17. Weaving as an instrumented automata product

A naive instrumented product adds an assignment and a guard for each event of the base program in the alphabet of the aspect but many optimizations are possible. In our example, it is easy to show that s is always equals to 1 in the initial state and equals to 2 in the second state. Tests and assignments are useless for the corresponding transitions; they can be suppressed (see Figure 17 (*b*)). In [20], we describe how to produce optimal (in terms of number of assignments) instrumentations. Actually, it is easy to show that assignments are only needed (yet not always) to distinguish between paths arriving at the same node. For a simple imperative base program, this involves adding at most an assignment for each conditional or loop statement.

The technique is easily adapted to timed automata. Figure 18 presents in-

strumented product with the same base program automaton as before and an availability aspect enforcing that each event a is followed by an event b after at most 5 time units (otherwise the program must be reset).

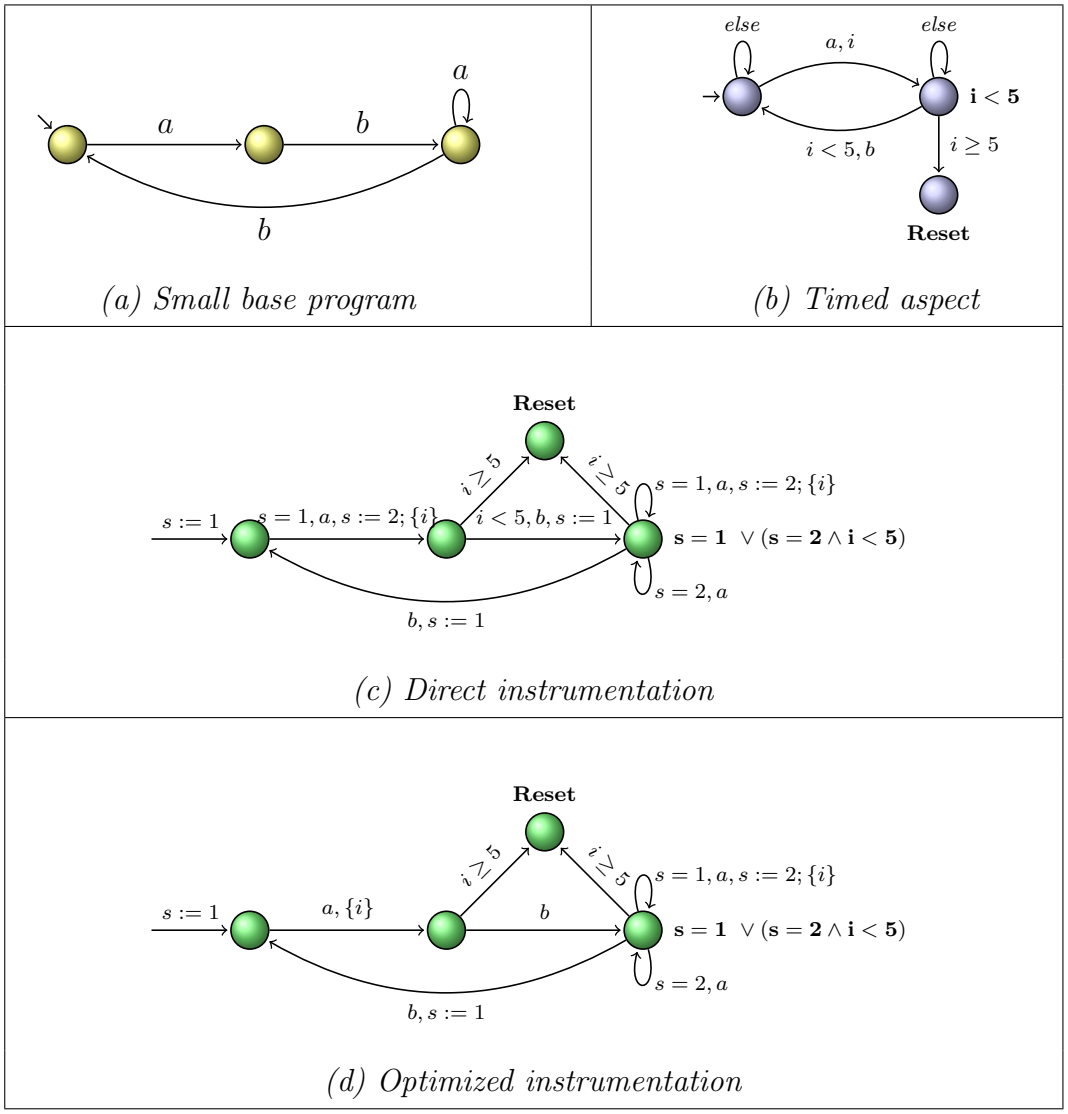

Fig. 18. Weaving as an instrumented timed automata product

The naive instrumented automata product is shown in Figure 18 (c). The encoding is similar as before. The underlying transition relation is encoded using guards and assignments on the state variable s. As a state may represent several states of the underlying product automaton, invariants can be disjunctions made of the invariants of the represented states.

The automaton, optimized using the same techniques as before, is shown in Figure 18 (d). The transition from the second state to **Reset** is useless since the action b occurs immediately after a. It will be removed by subsequent temporal optimizations (see Section 8)

The resulting automaton is nearly an UPPAAL automaton. The only departure is the disjunctive invariants which must be conjunction in UPPAAL. However, in our case, the disjunctions are mutually exclusive and it is very easy to rewrite this automaton into pure UPPAAL syntax before performing the verification step.

Constructing an instrumented product entails a linear code expansion in the worst case whereas a synchronized product may entail a quadratic blowup. The complexity remains the same with multiple aspects which can be represented by a single aspect (their product automaton) and woven, as before, using a single variable representing its state.

Of course, the standard or instrumented product automata represent the same automaton. The instrumentation is just a compact encoding that will not change the complexity of the verification step. The objective of this technique is only to prevent the production of too large programs. The size of a woven program will always remains close to the size of the corresponding base program. This benefit comes at the price of assigning and testing the variable representing the state of the aspect. Even if a small time overhead is preferable to a space explosion, there are also cases where the standard product is more appropriate. A possible extension would be to give the user the opportunity to specify on which automaton (or on which parts of an automaton) standard or instrumented product must be used.

Optimization, verification and concretization The optimization step is expressed as a product with small automata. It relies in part on a cost analysis which may be expensive to be precise. Again, many tradeoffs between cost and precision are possible. For example, it is possible to produce efficiently a cost function by assigning their WCET to basic blocks of instructions and $[0, +\infty]$ to those containing problematic constructs such as while loops, recursion, *etc.*. The verification step is potentially highly costly. Model checking timed properties (*e.g.*, TCTL) for timed automata is PSPACE-complete [1]. The instrumented product does not improve this step since all the encodings (using a bounded integer variable and disjunctive invariants) entails expansion of the model before or during the verification. Nevertheless, UPPAAL is able to verify properties of large systems. The last step, concretization, is a linear time traversal of the automaton.

To summarize, the verification is the only step whose cost may be prohibitive. However, this is an optional step in the weaving process. The main objective is to express resource management policies separately and to implement (weave) them automatically. Actually, with appropriate tradeoffs the weaving of availability aspects can be implemented in linear time.

10 Case study

Our case study is the program of an automatic teller machine (ATM) which is a standard software engineering example (see *e.g.,* [22,23]). An ATM usually includes several constraints about, for example, the duration or the number of tries to enter the PIN. The implementation of these constraints is usually scattered across the program and represents a typical crosscutting concern.

We show how our approach can simplify the implementation of these constraints by specifying them separately from the basic functionality. Some constraints are temporal and require the use of timers and waiting loops; others are untimed safety properties. Our approach permits to describe all of them as aspects. In that respect, the case study shows that our technique can be applied to general, timed or untimed, safety properties.

The base functionality of the `ATM` is defined by the following program:

$$\texttt{ATM} = \left\{ \begin{array}{ll} l_0 : \texttt{waitCard}() & \rightsquigarrow l_1 \\ l_1 : \texttt{pinPrompt}() & \rightsquigarrow l_2 \\ l_2 : \texttt{enterPin}() & \rightsquigarrow l_3 \\ l_3 : \texttt{checkPin}() & \rightsquigarrow l_4 \ ; \ l_1 \\ l_4 : \texttt{amountPrompt}() & \rightsquigarrow l_5 \\ l_5 : \texttt{enterAmount}() & \rightsquigarrow l_6 \\ l_6 : \texttt{checkAmount}() & \rightsquigarrow l_7 \ ; \ l_4 \\ l_7 : \texttt{cashCollection}() & \rightsquigarrow l_8 \\ l_8 : \texttt{cardReturn}() & \rightsquigarrow l_0 \end{array} \right\}$$

In its initial state, the ATM waits for a user to insert a card (`waitCard`()). Then, it prints a prompt asking for the PIN (`pinPrompt`()), waits for it (`enterPin`()) and checks it (`checkPin`()). If it is invalid, the ATM loops and asks for a new PIN. If the PIN is valid, it asks for the amount to withdraw (`amountPrompt`()), waits for it (`enterAmount`()) and checks it (`checkAmount`()). If it is invalid, the ATM loops and asks for a new amount. If the amount is valid, the ATM yields the corresponding cash (`cashCollection`()) and returns the card (`cardReturn`()).

We consider three different constraints concerning the time limit to enter the PIN, the number of tries allowed and the maximum duration of processing after a correct PIN is entered. They are described by the three following aspects.

Aspect A_1 specifies that the PIN has to be entered between 5 and 60 seconds after printing of the prompt.

$$A_1 = \left\{ \begin{array}{l} a_1 = \texttt{pinPrompt} \triangleright \{\mathit{reset}(i_1, 60), \mathit{start}(t_1)\};\ a_2 \\ a_2 = \texttt{enterPin} \triangleright \{\mathit{wait}(t_1, 5), \mathit{cancel}(i_1)\};\ a_1 \end{array} \right\}$$

After the instruction `pinPrompt`, an interrupt i_1 is programmed to be triggered 60 seconds later and the timer t_1 is started to enforce that `enterPin` is executed at least 5 seconds after `pinPrompt`. Then, the instruction `enterPin` cancels the interrupt i_1.

Aspect A_2 specifies that after entering three invalid PINs, the ATM must be reinitialized.

$$A_2 = \left\{ \begin{array}{l} a_1 = \overline{\texttt{checkPin}} \triangleright \mathit{nop};\ a_2 \\ a_1 = \texttt{checkPin} \triangleright \mathit{nop};\ a_1 \\ a_2 = \overline{\texttt{checkPin}} \triangleright \mathit{nop};\ a_3 \\ a_2 = \texttt{checkPin} \triangleright \mathit{nop};\ a_1 \\ a_3 = \overline{\texttt{checkPin}} \triangleright \mathit{reset}(i_2, 0);\ a_1 \\ a_3 = \texttt{checkPin} \triangleright \mathit{nop};\ a_1 \end{array} \right\}$$

The reinitialization is done using an immediate reset ($\mathit{reset}(i_2, 0)$).

Aspect A_3 imposes that the duration of processing after a correct PIN is entered may not exceed 180 seconds.

$$A_3 = \left\{ \begin{array}{l} a_1 = \texttt{checkPin} \triangleright \mathit{reset}(i_3, 180);\ a_2 \\ a_2 = \texttt{cardReturn} \triangleright \mathit{cancel}(i_3);\ a_1 \end{array} \right\}$$

In these aspects, the *reset* returns the card and reinitializes the ATM to its initial state waiting for a card. This is the correct behavior when time constraints are violated. In real life, some ATMs retain the card after three invalid PINs. A parameterized *reset* instruction would be sufficient to express these different kinds of reinitializations.

Not all the instructions of the base program are of interest for the aspects. The abstraction aggregates the instructions from l_4 to l_7 into a single composite one. In order to present a simpler automaton, we use the instruction `process` to represent the block of instructions from l_4 to l_8 (*i.e.*, including `cardReturn`). The aspect A_3 is rewritten to check that the duration between `checkPin` and the end of `process` does not exceed 180 seconds.

Figure 19 shows the timed automaton of the ATM obtained after abstraction, weaving and optimization. Aspects A_1 and A_3 are woven by a standard timed automata product. Since a standard product with aspect A_2 would duplicate nodes, we weave it using an instrumented product (see Section 9).

As in Section 8, we make use of timers (here u, k and seq) to take into account the execution time of instructions, the immediate sequencing of instructions and to take $\mathbb{B}()$ transitions at the earliest. We have considered that all the basic instructions of the ATM last one second except the instructions waiting for the user's input which have an unbounded execution time. That timing information permits to analyze the automaton to remove useless transitions. For example, the interruption i_1 of aspect A_1 can only be triggered in three states of the woven automaton.

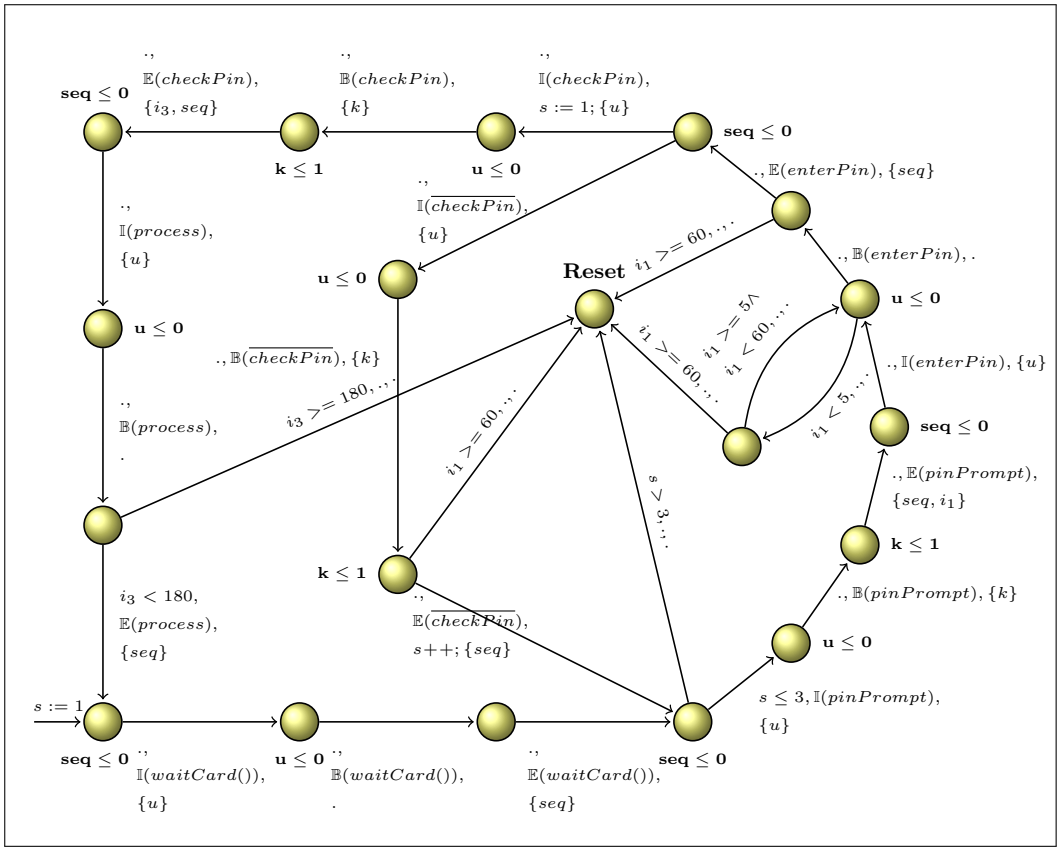

Fig. 19. Timed automaton of the `ATM` after weaving and optimization

The state of the automaton A_3 is encoded by the integer variable s which is initialized to 1. Transition $\mathbb{E}(\overline{checkPin})$ is annotated by `s++` which summarizes the state-transition table of the aspect A_2 (the state of A_2 changes each time an invalid PIN is entered). The program is reset after $\mathbb{E}(\overline{checkPin})$ if $s > 3$ *i.e.*, if more than three invalid PINs have been entered.

It can be verified that the complete process (from the insertion of the card

until it is returned) takes between 6 and 363 seconds (at worst, 3*61 seconds to enter a PIN and 180 seconds for the processing).

Concretization of the woven automaton yields the following program:

$$
\texttt{ATM} = \left\{ \begin{array}{lll}
l_0 & : s := \mathit{1} & \rightsquigarrow l_1 \\
l_1 & : \texttt{waitCard}() & \rightsquigarrow l_2 \\
l_2 & : s > \mathit{3} & \rightsquigarrow l_3 \ ; \ l_4 \\
l_3 & : \mathit{reset}(\texttt{i}_2, \texttt{0}) & \rightsquigarrow l_0 \\
l_4 & : \texttt{pinPrompt}() & \rightsquigarrow l_5 \\
l_5 & : \mathit{reset}(\texttt{i}_1, \texttt{60}) & \rightsquigarrow l_6 \\
l_6 & : \mathit{start}(\texttt{t}_1) & \rightsquigarrow l_7 \\
l_7 & : \mathit{wait}(\texttt{t}_1, \texttt{5}) & \rightsquigarrow l_8 \\
l_8 & : \mathit{cancel}(\texttt{i}_1) & \rightsquigarrow l_9 \\
l_9 & : \texttt{enterPin}() & \rightsquigarrow l_{10} \\
l_{10} & : \texttt{checkPin}() & \rightsquigarrow l_{13} \ ; \ l_{11} \\
l_{11} & : s{+}{+} & \rightsquigarrow l_2 \\
l_{12} & : \mathit{reset}(\texttt{i}_3, \texttt{180}) & \rightsquigarrow l_{13} \\
l_{13} & : \texttt{amountPrompt}() & \rightsquigarrow l_{14} \\
l_{14} & : \texttt{enterAmount}() & \rightsquigarrow l_{15} \\
l_{15} & : \texttt{checkAmount}() & \rightsquigarrow l_{16} \ ; \ l_{13} \\
l_{16} & : \texttt{cashCollection}() & \rightsquigarrow l_{17} \\
l_{17} & : \texttt{cardReturn}() & \rightsquigarrow l_{18} \\
l_{18} & : \mathit{cancel}(\texttt{i}_3) & \rightsquigarrow l_1
\end{array} \right\}
$$

Usually, the specification of an ATM does not separate the basic functionality from error processing. Our approach makes the separation of these concerns possible.

11 Related work

Yu and Gligor [10] present a method to verify that a resource allocator remains available. In their framework, a denial of service is defined as a scenario

between one or several users preventing some others to access a resource.

They consider systems composed of users and resources. Users are specified by sequences of requests. Resources are formally specified by:

- a collection of access operations. For example, `mutex` resources involve the operations `take` and `release`;
- an internal state (a set of variables). A `mutex` resource has a boolean indicating whether the resource is allocated or free as well as a variable recording the identity of the user owning the resource;
- properties specifying the proper use of the resource. Such properties might be, for example, that users' requests are processed one at a time in a FIFO ordering.

Yu and Gligor's model considers finite time availability properties. Typically, properties express that requests will be processed in a finite (but not necessarily bounded) amount of time. Bounded time availability properties are stronger constraints. Yu and Gligor make clear that some constraints must be enforced on users to guarantee the availability of resources. *User agreements* define properties (*e.g.,* liveness) that users must satisfy. For example, a user performing a request `take` on a `mutex` resource must eventually perform a `release`. Their methodology allows to prove that the system specification and user agreements guarantee the expected availability properties. For a `mutex` resource, it consist in showing that all `take` and `release` requests are eventually performed. Availability is guaranteed if user agreements ensure that any `take` is eventually followed by a `release`.

Millen [24] considers other availability policies based on bounded (the resource will be granted before a given delay) or probabilistic (the resource will be granted according a probabilistic law) waiting time. His model relies on a global monitor managing all resource accesses. The monitor uses a *Denial of service protection base* (DPB) extending the standard *Trusted Computing Base* (TCB). A DPB ensures that all requests are performed through the monitor and that resources can be released by the monitor. Ensuring the access to resources is not sufficient to prevent every denial of service. Indeed, the resource can be released by the monitor before the user is done. Users should specify the resource they need but also the time they need to retain it.

Our framework can be seen as an adaptation of Yu and Gligor's framework to bounded time policies. In particular, our services can be seen as their users and aspects as user agreements. In both cases, availability properties can be proved by taking into account the specification of the system and aspects/user agreements. Of course, the major difference is the use of aspects which allows a better separation of concerns and, above all, an automatic instrumentation of programs using weaving. Like Millen's monitors, we consider bounded time

properties. However, we make use of local properties that are designed for and woven to each service. Local policies often suffice to ensure availability properties. They are also easier to design, understand and implement efficiently.

Cuppens and Saurel [25] introduce a framework based on temporal and deontic logics to specify availability policies. They can verify the internal consistency of availability policies, whether a policy ensures specific and required availability properties or if a (logical specification of a) system satisfies an availability policy. Their approach is suitable to verify policies *a posteriori* but not to enforce them.

J-Seal2 [26] is a secure mobile agent system proposing a simple and global mechanism to ensure availability of processors and memory. The system provides resource control to limit the usage of physical resources like CPU and of logical resources like threads. Their main goal is a completely portable implementation of resource control. It is described in terms of code instrumentation but it is not generic enough to be used for other types of resources (*e.g.*, resources with exclusive access).

Nandivada and Palsberg [27] abstract a TCP server into a timed automaton. A WCET analysis is performed on the intermediate RTL code produced by gcc. They focus on flooding attacks which are also represented as timed automata. UPPAAL is applied to the whole system (server and attacker) to verify the ability of the TCP server to survive denial-of-service attacks. They do not consider the enforcement of availability properties but we could reuse their timing analysis to abstract our services and infer time information.

Several AOP-related approaches also rely on automata. Let us mention:

- Ligatti, Bauer and Walker [28] who introduce edit automata which may terminate programs as well as suppress or insert sequences of actions. These automata are used to implement security monitors and enforce safety properties;
- Sipma [29] represents aspects as transformations of transition systems. That framework is used to formally analyze common aspect constructs.
- Altisen, Maraninchi and Stauch [30] investigate the use of AOP for reactive languages. They propose a dedicated aspect language and prove that weaving preserves the usual behavioral equivalence for reactive systems.

Their respective goals and techniques are quite different from ours; in particular, none of them consider timed properties and automata.

12 Conclusion

We have proposed a formal framework to enforce availability properties on services sharing resources. At a practical level, we have defined a domain-specific aspect language dedicated to the prevention of denial of service. At a methodological level, our approach promotes a formal view of AOP with aspects as properties and weaving as an automata product.

We have shown in [7] the correctness of the whole approach (abstraction, weaving, concretization) in a simpler (untimed) setting. We have shown that if a program respects the aspect (a safety property) then the woven program has the same behavior. If a program does not respect the aspect then the woven program is stopped just before the violation. With availability aspects, proofs need to refer to the timed semantics of services. We have not completed that generalization yet but we believe that the structure of the proofs remains identical.

The implementation of our technique is likely to be realistic. The representations of services should remain of moderate size since code unrelated to resource management can be represented by a single instruction. The costs of analyses (control flow, execution time) can be controlled by adjusting the precision of their approximation. Finally, if a weaving based on a standard automata product may involve a code explosion in some cases, it is easy to circumvent this problem by replacing code duplication by code instrumentation (see [20]).

This research belongs to a series of work considering aspects as formal properties on execution traces. The joint technique is to translate programs and aspects into (various forms of) automata and to express weaving as a kind of automata product.

- In [20], we have proposed a technique to enforce user-defined security policies expressed as automata. A potential use of the method is the securing of applets using a just-in-time weaving of the policies/aspects. The instrumentation performed by weaving ensures that the applet will be stopped just before it tries to infringe the policy.
- In [19], we have proposed domain-specific aspects to specify and enforce scheduling policies to networks of communicating processes. A scheduling aspect (expressed as an automaton) selects a subset of allowed execution traces of the set of all possible interleavings. This technique permits transformation of a network into an equivalent (and more efficient) sequential program.
- In this article, we have generalized our previous framework to timed automata in order to express and enforce properties on execution time. We

can prevent some execution traces and also modify their timed behavior. Our aspect language is expressive enough to specify many different availability policies.

That series shares the same goal of keeping the semantic impact of weaving under control in order to permit reasoning (analyses, verification, proofs) on aspect-oriented programs. In general purpose aspect languages with unrestricted advice, it is very difficult, in general, to predict the effect of weaving and to reason compositionally.

We are currently completing the formalization of the concretization and the associated correctness proofs. A useful extension would be to provide better support for the prevention of deadlocks. Limiting the duration of resource allocation or enforcing an allocation ordering (*cf.* Section 6.2) permits avoidance of deadlocks. However, these techniques are not always satisfactory. The system can often be stuck waiting for a time limit to be reached. Worse, a bad allocation ordering may involve systematic interrupts of services which will not be able to perform their task anymore. A better solution would be to transform services such that they allocate some resources earlier (but therefore longer) to satisfy the allocation ordering specified by the aspect. We have not formalized this transformation but it seems that static analyses techniques would be useful to find the best timing satisfying the allocation ordering. Another option would be to specify global policies for deadlock prevention. Shared variables representing the availability of resources could be used to schedule their allocation to services. Using such information, an aspect could state, for example, that allocating a resource is not possible if another resource is already allocated to another service.

Another interesting research direction would be to model in our framework more sophisticated availability policies relying, for example, on dynamic performance evaluation, admission control or priorities.

Acknowledgements

This work has been supported by the ACI Dispo project.

References

[1] R. Alur, D. L. Dill, A theory of timed automata, Theoretical Computer Science 126 (2) (1994) 183–235.

[2] K. G. Larsen, P. Pettersson, W. Yi, UPPAAL in a nutshell, Int. J. on Software Tools for Technology Transfer 1 (1-2) (1997) 134–152.

[3] J. Bengtsson, W. Yi, Timed automata: Semantics, algorithms and tools, in: Lectures on Concurrency and Petri Nets, LNCS vol. 3098, Springer, 2003, pp. 87–124.

[4] P. Fradet, S. Hong Tuan Ha, Aspects of availability, in: Proc. of the sixth international conference on Generative Programming and Component Engineering (GPCE'07), ACM Press, 2007, pp. 165–174.

[5] P. Fradet, S. Hong Tuan Ha, Systèmes de gestion de ressources et aspects de disponibilité, in: 2^e Journée sur le Développement de Logiciels Par Aspects (JFDLPA 2005), 2005.

[6] P. Fradet, S. Hong Tuan Ha, Systèmes de gestion de ressources et aspects de disponibilité, L'Objet - Logiciel, bases de données, réseaux 12 (2-3) (2006) 183–210.

[7] S. Hong Tuan Ha, Programmation par aspects et tissage de propriétés. application à l'ordonnancement et à la disponibilité., Ph.D. thesis, Rennes University (Jan. 2007).

[8] J.-C. Laprie, Dependability: Basic Concepts and Terminology, Dependable Computing and Fault-Tolerant Systems, Springer, 1992.

[9] J. Rushby, Critical system properties: Survey and taxonomy, Reliability Engineering and Systems Safety 43 (2) (1994) 189–219.

[10] C.-F. Yu, V. D. Gligor, A specification and verification method for preventing denial of service, IEEE Trans. Soft. Eng. 16 (6) (1990) 581–592.

[11] R. Alur, Timed automata, in: 11th International Conference on Computer Aided Verification, LNCS vol. 1633, Springer–Verlag, 1999, pp. 8–22.

[12] J. Leiwo, Y. Zheng, A method to implement a denial of service protection base, in: ACISP '97: Proceedings of the Second Australasian Conference on Information Security and Privacy, Springer-Verlag, 1997, pp. 90–101.

[13] F. B. Schneider, Enforceable security policies, ACM Transactions on Information and System Security 3 (1) (2000) 1–50.

[14] R. Douence, P. Fradet, M. Südholt, A framework for the detection and resolution of aspect interactions, in: Proc. of Conference on Generative Programming and Component Engineering (GPCE'02), LNCS vol. 2487, Springer–Verlag, 2002, pp. 173–188.

[15] R. Douence, P. Fradet, M. Südholt, Trace-based aspects, in: M. Aksit, S. Clarke, T. Elrad, R. Filman (Eds.), Aspect-Oriented Software Development, Addison-Wesley, 2004, pp. 201–217.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An overview of AspectJ, Lecture Notes in Computer Science 2072 (2001) 327–355.

[17] P. Puschner, C. Koza, Calculating the maximum, execution time of real-time programs, Real-Time Syst. 1 (2) (1989) 159–176.

[18] X. Li, T. Mitra, A. Roychoudhury, Modeling control speculation for timing analysis, Real-Time Syst. 29 (1) (2005) 27–58.

[19] P. Fradet, S. Hong Tuan Ha, Network fusion, in: Prog. Lang. and Syst.: Second Asian Symposium, (APLAS'04), LNCS vol. 3302, 2004, pp. 21–40.

[20] T. Colcombet, P. Fradet, Enforcing trace properties by program transformation, in: Symposium on Principles of Programming Languages (POPL'00), 2000, pp. 54–66.

[21] D. V. Horn, H. G. Mairson, Relating complexity and precision in control flow analysis, in: ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming, 2007, pp. 85–96.

[22] S. Somé, R. Dssouli, J. Vaucher, From scenarios to timed automata: building specifications from users requirements, in: Asia Pacific Software Engineering Conference, 1995, pp. 48–57.

[23] B. Regnell, K. Kimbler, A. Wesslen, Improving the use case driven approach to requirements engineering, in: IEEE International Conference on Requirements Engineering, 1995, pp. 40–48.

[24] J. K. Millen, A resource allocation model for denial of service protection, Journal of Computer Security 2 (2-3) (1993) 89–106.

[25] F. Cuppens, C. Saurel, Towards a formalization of availability and denial of service, in: Inf. Syst. Tech. Panel Symp. on Protecting Nato Information Systems in the 21st century, 1999.

[26] W. Binder, J. G. Hulaas, A. Villaz, Portable resource control in Java, in: OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, ACM Press, 2001, pp. 139–155.

[27] V. K. Nandivada, J. Palsberg, Timing analysis of TCP servers for surviving denial-of-service attacks., in: IEEE Real-Time and Embedded Technology and Applications Symp., 2005, pp. 541–549.

[28] J. Ligatti, L. Bauer, D. Walker, Edit automata: enforcement mechanisms for run-time security policies, Int. J. Inf. Security 4 (1-2) (2005) 2–16.

[29] H. Sipma, A formal model for cross-cutting modular transition systems, in: Workshop on Foundations of Aspect-Oriented Languages (FOAL'03), 2003.

[30] K. Altisen, F. Maraninchi, D. Stauch, Aspect-oriented programming for reactive systems: Larissa, a proposal in the synchronous framework., Sci. Comput. Program. 63 (3) (2006) 297–320.