Aspects of Availability

Pascal Fradet [*]
INRIA / LIG
Pascal.Fradet@inria.fr

Stéphane Hong Tuan Ha [†]
INRIA / CEA-LIST
Stephane.Hong-Tuan-Ha@cea.fr

Abstract

In this paper, we propose a domain-specific aspect language to prevent the denials 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.

Categories and Subject Descriptors D.2.3 [*Coding Tools and Techniques*]: Structured programming; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]; I.2.2 [*Automatic Programming*]: Program transformation

General Terms Languages, Reliability, Security, Verification

Keywords 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 authorized subjects are answered in a timely manner. In other words, there is no *denial of service*. In this paper, we study resource management in isolation (*i.e.*, separately from the basic functionality) and address the prevention of denials of services (*i.e.*, availability) using aspect-oriented techniques. We propose a domain-specific aspect language in order to prevent denials 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.

[*] INRIA Rhône-Alpes, 655, av. de l'Europe, 38330 Montbonnot, France

[†] CEA Saclay, DRT/LIST/DTSI/LSL, 91191 Gif sur Yvette Cedex, France

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* [3]. 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. 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 more 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 [16, 4]) 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. In section 3, we briefly recall the main characteristics of timed automata. Sections 4 and 5 present the syntax and semantics of services and availability aspects, respectively. The technical core of the paper lies in section 6 which describes the abstraction of services and semantics of aspects in terms of timed automata and the weaving as an automata product. The optimization, verification and concretization of the final (woven) automaton back into a source program are sketched in section 7. We conclude by presenting related work and possible extensions in section 8.

Previous versions of this work have been published in a French conference [11], journal [12] and PhD thesis [13]. Correctness proofs of our approach in a simpler setting can be found in [13].

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 which 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 file; 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.

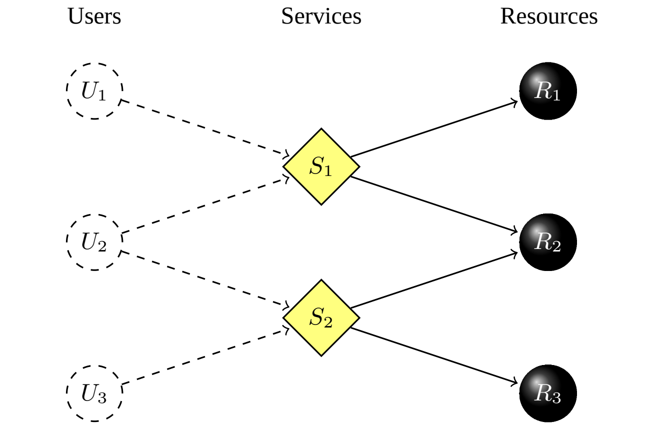


Figure 1. Three-layer model

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 their requests any further. Since we are concerned by resource management and the prevention of denial of services, we focus on interactions between services and resources.

The availability problems we consider come from concurrent accesses of services 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 denials of service must be addressed by dedicated fault-tolerance techniques (see for example [15, 22]).

2.2 Approach

Yu and Gligor have studied denials of service caused by resource management [25]. They have shown that availability properties depend 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 [11].
- 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 5).

In this paper, we 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 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

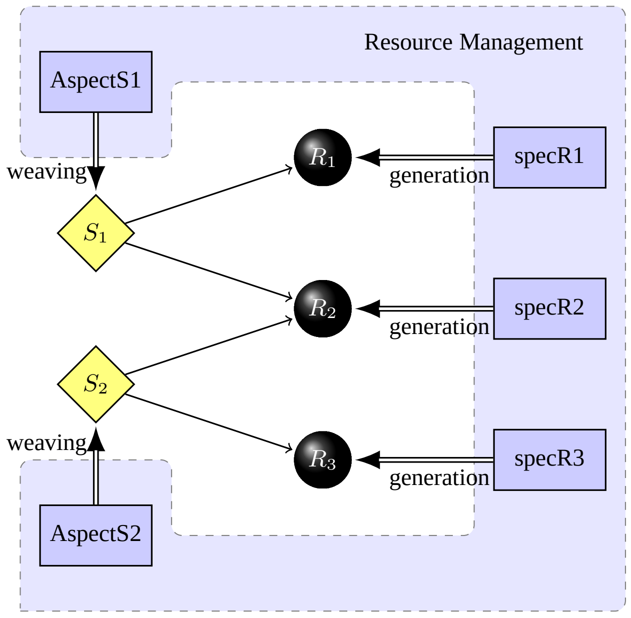


Figure 2. Global layout of the system

the complete system. Local aspects are sufficiently expressive to prevent many denials 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 6.1);
- an aspect is defined using a domain specific language. Its semantics is given by a timed automaton (section 6.2);
- 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 6.3);
- 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 7.1);
- it is possible to automatically verify that the woven automaton satisfies general availability properties (section 7.2);
- the last step amounts to concretizing the (optimized and verified) automaton into source code using timed commands (watchdog timers, waiting loops, interrupts) (section 7.3).

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 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:

Services

```
       ⎧ l1 :  M1.alloc();
       ⎪       M2.alloc();
S1 =   ⎨       S1Comput;
       ⎪       M2.free();
       ⎪       M1.free();
       ⎩       jmp l1;

       ⎧ l2 :  M2.alloc();
       ⎪       S2Comput1;
       ⎪       if G then
S2 =   ⎨          M1.alloc();
       ⎪          S2Comput2;
       ⎪          M1.free();
       ⎪       M2.free();
       ⎩       jmp l2;
```

Network

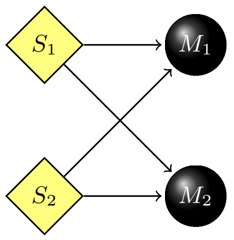


Figure 3. A simple system with two services and two resources

- 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, 16] 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

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

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 $(Q, q_0, H, \Sigma, \delta, I)$ 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 denoting actions of the automaton;
- $\delta \subseteq Q \times 2^C \times \Sigma \times 2^H \times Q$ is the transition relation;
- $I : Q \to 2^C$ maps a state to its *invariant*.

A transition $(q, g, a, r, q') \in \delta$ 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, $q \xrightarrow{.,.,.} q'$ denotes the spontaneous transition.

3.2 Semantics

The operational semantics of a timed automaton $(Q, q_0, H, \Sigma, \delta, I)$ 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 $u : H \to \mathbb{R}$ 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 uses the following notations:

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

The transitions are either transitions representing the time passing

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

or transitions representing the execution of an action

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

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.

3.3 Timed automata product

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

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

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

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

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 the two constitutive 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 constitutive guards and 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.

4. Services

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

4.1 Syntax

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

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

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 as such).

That source language is very simple yet sufficiently expressive. Its main advantage is that source programs are very close to their control flow graph which will be translated to a timed automaton. A higher-level language could be considered using a control flow graph 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:

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

The commands getUser(), 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.

4.2 Semantics

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

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

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

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

5. 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 [23]. Since only safety properties can be enforced by weaving, we consider *bounded time* 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.

5.1 Syntax

Our language is inspired by *stateful aspects* [8] (or trace-based aspects [9]) which take the history of execution into account. The syntax is described in figure 4.

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

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

Figure 4. Syntax of availability aspects

An aspect is a collection of mutually recursive equations. An equation is of the form $a_i = (C \rhd L); a_j$ and should be read as: the aspect waits for the event C which triggers the execution of a sequence of instructions L and passes the control to equation a_j. In general, an equation may contain choices. For example, the aspect $(C \rhd L); a \,\Box\, (C' \rhd L'); a'$ 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 [14], is either a simple pattern (a term, possibly with wildcards *, matching instructions), 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.

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

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 (*wait*$(t, k) \equiv nop$);
- 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 which 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.

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

5.2 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:

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

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

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 reallocating 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 M as soon as it has released it whereas Y asks for it 20 seconds after it has started to process a new request. Better fairness can be guaranteed by making the service X wait at least 20 seconds between each allocation of M. The following aspect specifies such a property:

$$\left\{ \begin{array}{lll} a_1 & = & \texttt{M.alloc} \triangleright start(t);\ a_2 \\ a_2 & = & \texttt{M.alloc} \triangleright \{\textit{wait}(t, 20); start(t)\};\ a_2 \end{array} \right\}$$

As soon as the event M.alloc is executed, a timer t is started. A wait of at least 20 seconds is imposed before a new event M.alloc is performed (*wait*$(t, 20)$). Afterward, the timer is reset and restarted.

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:

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

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.

$$\left\{ \begin{array}{lll} a_1 & = & \texttt{M2.alloc} \triangleright \{\};\ a_2 \\ a_2 & = & \texttt{M1.alloc} \triangleright \textit{reset}(i, 0);\ a_3 \\ & \Box & \texttt{M2.free} \triangleright \{\};\ a_1 \end{array} \right\}$$

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.

6. 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 constraint (*i.e.,* all timing behaviors are included). Then, we give the semantics of aspects in terms of timed automata. The next step is to weave the aspect on 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. A last step implements the waiting constraints by the instruction *wait*.

6.1 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 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:

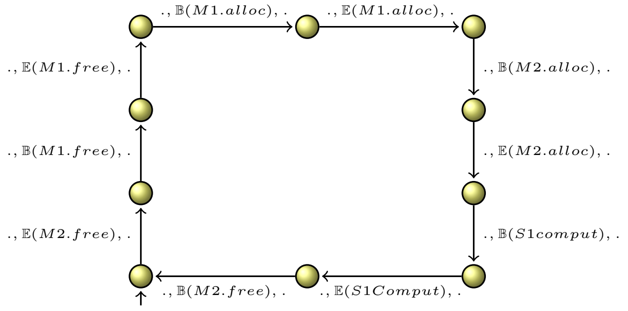
$$\mathbf{next}_S(l_1, a, l_2) \textit{ iff}$$
$$l_1 : a \rightsquigarrow l_2 \in S \;\vee\; l_1 : a \rightsquigarrow l_2 \,;\, l \in S \;\vee\; l_1 : \neg a \rightsquigarrow l \,;\, l_2 \in S$$

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:
$$\Sigma_{S^\sharp} = \{l, l_a \;\mid\; \mathbf{next}_S(l, a, l')\}$$
- 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 two actions (instants) $\mathbb{B}(a)$ (the beginning of a) and $\mathbb{E}(a)$ (the end of a):
$$\mathcal{E}_{S^\sharp} = \{\mathbb{B}(a), \mathbb{E}(a) \;\mid\; a \in \mathcal{E}_S\}$$
Splitting the action in two instants is used to represent the execution time of actions ;
- the transition relation $\longrightarrow_{S^\sharp}$ is defined as follows:
$$(l, ., \mathbb{B}(a), ., l_a) \in \longrightarrow_{S^\sharp} \quad \wedge \quad (l_a, ., \mathbb{E}(a), ., l') \in \longrightarrow_{S^\sharp}$$
$$\textit{iff} \;\; \mathbf{next}_S(l, a, l')$$
Each action a from one state to another is represented using an intermediate state l_a and two transitions corresponding to the two instants $\mathbb{B}(a)$ and $\mathbb{E}(a)$ without any timing constraint (no constraints on the time passing between two actions).
- 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 5 illustrates the abstraction of service S1 into a timed automaton.



$., \mathbb{B}(M1.alloc), .$
$., \mathbb{E}(M1.alloc), .$
$., \mathbb{E}(M1.free), .$
$., \mathbb{B}(M2.alloc), .$
$., \mathbb{B}(M1.free), .$
$., \mathbb{E}(M2.alloc), .$
$., \mathbb{E}(M2.free), .$
$., \mathbb{B}(S1comput), .$
$., \mathbb{B}(M2.free), .$
$., \mathbb{E}(S1Comput), .$

Figure 5. 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:*

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

6.2 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.

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 and aspect;
- 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:
$$I_a(q, e) = \{i \le e(i) \;\mid\; \forall i.e(i) \ne \perp\}$$

In the remaining, 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:

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

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.

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

A rule $(F, G) \rhd L$ 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.

$$\begin{aligned}
&[(F, G) \rhd L]^{(q_1, e_1)}_{(q_2, e_2)} \\
&= \;\; \{\, (q_1, e_1) \xrightarrow{g,\mathbb{B}(a),.}_a (q', e_1) \\
&\qquad \cup\, (q', e_1) \xrightarrow{\bigwedge(e),\mathbb{E}(a),r}_a (q_2, e_2) \;\cup\; interrupt(q', e_1) \\
&\qquad \mid\; match(a, F) \;\wedge\; ins(e_1, L) = (g_i, r, e_2) \\
&\qquad\quad \wedge\; (g = \bigwedge(e_1) \cup G \cup g_i)\}
\end{aligned}$$

For each action a matched by F, two transitions ($\mathbb{B}(a)$ and $\mathbb{E}(a)$) are added using a new intermediate state (q', e_1). Transitions modeling interrupts are added to this state. The function ins analyzes the advice L to compute the guards and resets of transitions as well as the new interrupt environment.

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.
$$interrupt(q, e) = \{(q, e) \xrightarrow{i \ge e(i),.,.}_a \text{RESET} \mid e(i) \ne \perp\}$$
There is a transition to RESET each time an interrupt i reaches its trigger value recorded in the environment e.
- The function $match(a, F)$ 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.

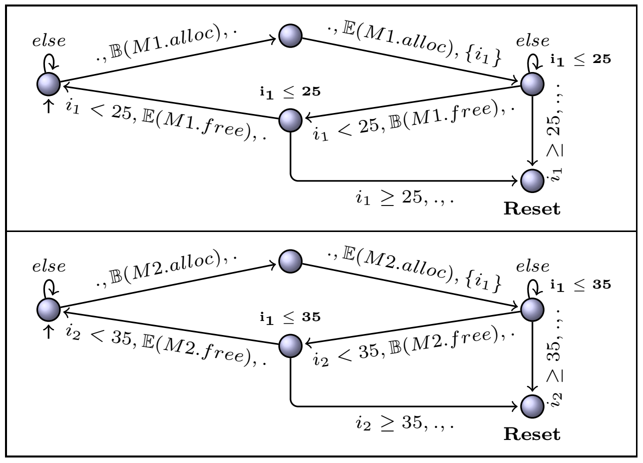
$$\begin{array}{rcl} ins(e, L) & = & (\{\mathbf{w}(t,k) \mid wait(t,k) \in L\}, \\ & & \{z \mid reset(z,k) \in L \vee start(z) \in L\}, \\ & & e') \end{array}$$

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

The *wait*(t,k) advice is represented by a special guard $\mathbf{w}(t,k)$. At the end of weaving, these guards are implemented by new states in which waiting is possible. Without this representation, we would need to add a new state before each instruction to represent possible waits.

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

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.



else
$., \mathbb{B}(M1.alloc), .$
$., \mathbb{E}(M1.alloc), \{i_1\}$
else
$\mathbf{i_1} \leq \mathbf{25}$
$\mathbf{i_1} \leq \mathbf{25}$
$i_1 < 25, \mathbb{E}(M1.free), .$
$i_1 < 25, \mathbb{B}(M1.free), .$
$i_1 \geq 25, ., .$
$i_1 \geq 25, ., .$
Reset
else
$., \mathbb{B}(M2.alloc), .$
$., \mathbb{E}(M2.alloc), \{i_1\}$
else
$\mathbf{i_1} \leq \mathbf{35}$
$\mathbf{i_1} \leq \mathbf{35}$
$i_2 < 35, \mathbb{E}(M2.free), .$
$i_2 < 35, \mathbb{B}(M2.free), .$
$i_2 \geq 35, ., .$
$i_2 \geq 35, ., .$
Reset

Figure 6. Timed automata of A_1 (above) and A_2 (below)

Figure 6 shows the semantic automata for the previously defined aspects A_1 and A_2. Intuitively, weaving will amount to starting a timer when the service takes the resource and to reset it when the timer reaches its time limit (*i.e.*, 25 or 35 seconds). 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.

6.3 Weaving an aspect to a service

Weaving *per se* is just the product (as described in section 3.3) of the automata representing the service and the aspect. 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.

A last step implements guards $\mathbf{w}(t,k)$ by *wait* instructions. More precisely, a transition of the form

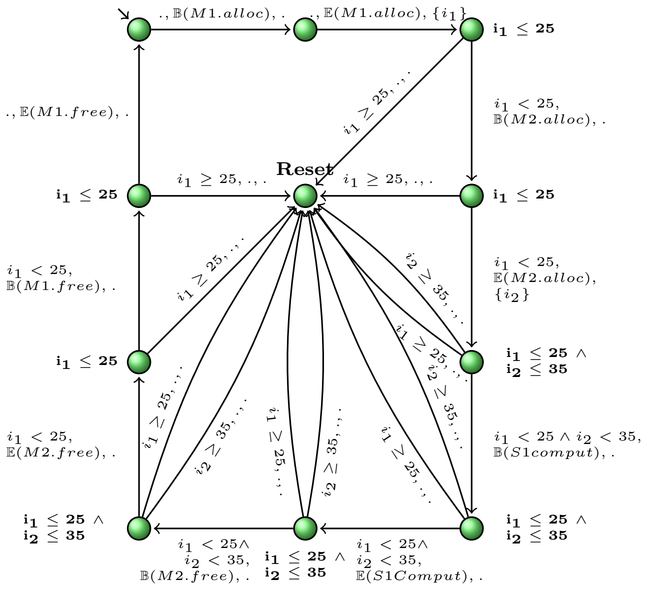
$$q \xrightarrow{\mathbf{w}(t,k) \wedge g, \mathbb{B}(c), .} q'$$

is translated into the following transitions

$$q \xrightarrow{(t \geq k) \wedge g, \mathbb{B}(c), .} q' \quad q \xrightarrow{(t < k) \wedge g, \mathbb{B}(wait), .} q_w \quad q_w \xrightarrow{(t \geq k), \mathbb{E}(wait), .} q$$

and the state invariant $t \leq k$ on state q_w. That is to say, if $t \geq k$ the action c can start otherwise the automaton performs a *wait* instruction *i.e.*, goes into the new state q_w where time passes until $t \geq k$.

Figure 7 shows the product of the abstraction of service `S1` with the aspects A_1 and A_2. In the product automaton, two interrupts are programmed after `M1.alloc` and after `M2.alloc`. If `M1.free` (resp. `M2.free`) is not executed before 25 seconds (resp. 35 seconds), the automaton goes to state RESET.



$., \mathbb{B}(M1.alloc), .$
$., \mathbb{E}(M1.alloc), \{i_1\}$
$\mathbf{i_1} \leq \mathbf{25}$
$., \mathbb{E}(M1.free), .$
$i_1 \geq 25, ., .$
$i_1 < 25,$ $\mathbb{B}(M2.alloc), .$
Reset
$\mathbf{i_1} \leq \mathbf{25}$
$i_1 \geq 25, ., .$
$i_1 \geq 25, ., .$
$\mathbf{i_1} \leq \mathbf{25}$
$i_1 < 25,$ $\mathbb{B}(M1.free), .$
$i_1 \geq 25, ., .$
$i_2 \geq 35, ., .$
$i_1 < 25,$ $\mathbb{E}(M2.alloc),$ $\{i_2\}$
$\mathbf{i_1} \leq \mathbf{25}$
$i_1 \geq 25, ., .$
$\mathbf{i_1} \leq \mathbf{25} \wedge$ $\mathbf{i_2} \leq \mathbf{35}$
$i_2 \geq 35, ., .$
$i_1 < 25,$ $\mathbb{E}(M2.free), .$
$i_1 \geq 25, ., .$
$i_2 \geq 35, ., .$
$i_1 \geq 25, ., .$
$i_2 \geq 35, ., .$
$i_1 \geq 25, ., .$
$i_1 < 25 \wedge i_2 < 35,$ $\mathbb{B}(S1comput), .$
$\mathbf{i_1} \leq \mathbf{25} \wedge$ $\mathbf{i_2} \leq \mathbf{35}$
$i_1 < 25 \wedge$ $i_2 < 35,$ $\mathbb{B}(M2.free), .$
$\mathbf{i_1} \leq \mathbf{25} \wedge$ $\mathbf{i_2} \leq \mathbf{35}$
$i_1 < 25 \wedge$ $i_2 < 35,$ $\mathbb{E}(S1Comput), .$
$\mathbf{i_1} \leq \mathbf{25} \wedge$ $\mathbf{i_2} \leq \mathbf{35}$

Figure 7. Product of service `S1` with aspects A_1 and A_2

In comparison with figure 5, transitions to RESET and state invariants have been added to model interrupts. Loops on the instruction *wait* have been suppressed since the aspects do not add delays.

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.

7. Optimization, verification and concretization

The product (woven) automaton can then be

- optimized by taking into account (worst-case and best case) execution times of instructions;

- 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.

7.1 Optimizations

We describe here how to optimize the woven automaton by taking into account time information. 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 [21, 17]) 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`:

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

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

  $$c_0 \xrightarrow{.,\mathbb{B}(a),\{k\}}_c q \quad \text{and} \quad q \xrightarrow{k \geq \text{BCET}(a),\mathbb{E}(a),.}_c c_0$$
  with q a fresh state

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

$$I_c(q) = \{k \leq \text{WCET}(a) \mid q \xrightarrow{k \geq \text{BCET}(a),\mathbb{E}(a),.}_c c_0\}$$

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 $k = \text{WCET}(a)$.

Another issue to take into account is that sequencing (*i.e.,* the ; operator) takes no time. In our example, 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 an action which may take time.

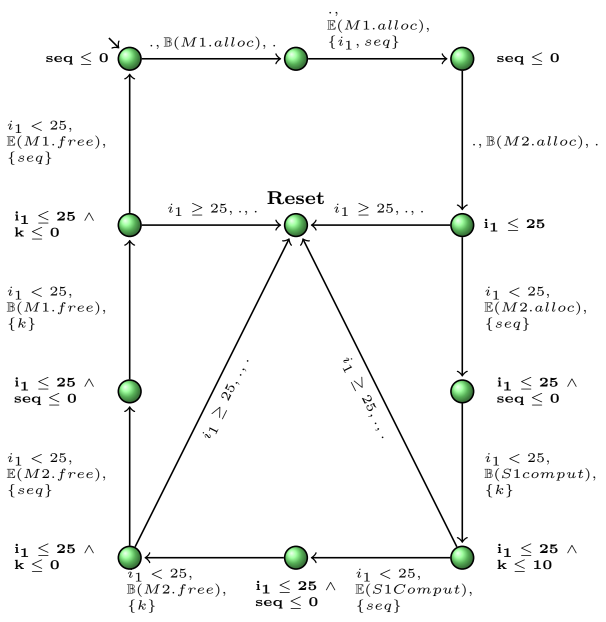
$$\longrightarrow_e = \left\{ \begin{array}{c} e_0 \xrightarrow{.,\mathbb{B}(a),.}_e e_1 \\ e_1 \xrightarrow{.,\mathbb{E}(a),\{seq\}}_e e_0 \end{array} \middle| \; \mathbb{B}(a) \in \mathcal{E}_{S\sharp} \wedge \mathbb{E}(a) \in \mathcal{E}_{S\sharp} \right\}$$

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

$$I_e(e_0) = \{seq \leq 0\} \qquad I_e(e_1) = \emptyset$$

The timed automaton obtained after the product with the cost and sequencing automata is more precise. The two products 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 8 shows the service `S1` of figure 7 after product with the sequencing and cost automaton corresponding to f_{cost} and simplification.



$\mathbf{seq \leq 0}$
$., \mathbb{B}(M1.alloc), .$
$.,$ $\mathbb{E}(M1.alloc),$ $\{i_1, seq\}$
$\mathbf{seq \leq 0}$
$i_1 < 25,$ $\mathbb{E}(M1.free),$ $\{seq\}$
$., \mathbb{B}(M2.alloc), .$
Reset
$i_1 \geq 25, ., .$
$i_1 \geq 25, ., .$
$\mathbf{i_1 \leq 25 \wedge}$ $\mathbf{k \leq 0}$
$\mathbf{i_1 \leq 25}$
$i_1 < 25,$ $\mathbb{B}(M1.free),$ $\{k\}$
$i_1 < 25,$ $\mathbb{E}(M2.alloc),$ $\{seq\}$
$\mathbf{i_1 \leq 25 \wedge}$ $\mathbf{seq \leq 0}$
$i_1 \geq 25, ., .$
$i_1 \geq 25, ., .$
$\mathbf{i_1 \leq 25 \wedge}$ $\mathbf{seq \leq 0}$
$i_1 < 25,$ $\mathbb{E}(M2.free),$ $\{seq\}$
$i_1 < 25,$ $\mathbb{B}(S1comput),$ $\{k\}$
$\mathbf{i_1 \leq 25 \wedge}$ $\mathbf{k \leq 0}$
$\mathbf{i_1 \leq 25 \wedge}$ $\mathbf{k \leq 10}$
$i_1 < 25,$ $\mathbb{B}(M2.free),$ $\{k\}$
$\mathbf{i_1 \leq 25 \wedge}$ $\mathbf{seq \leq 0}$
$i_1 < 25,$ $\mathbb{E}(S1Comput),$ $\{seq\}$

Figure 8. Timed automaton of `S1` after weaving and optimization

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.

7.2 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 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 and A_2 on services `S1` and `S2` and written the result with UPPAAL. We have verified that the woven system respected 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* 45 *seconds*. This property can be verified using a new timer `avail` reset at the beginning of the request loop and by checking `avail`≤ 45 for all states. This property has been ensured by weaving. Indeed, the woven service S2 must release resource M2 at most after 35 seconds, so S1 cannot wait more than 35 seconds to get its resources. Since S1Comput takes at most 10 seconds, S1 will terminate before 45 seconds. This also means that service S1 will always get access to the needed resources and, more generally, that no denial of resources can arise in the system 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.

7.3 Concretization

The concretization of a standard automaton into our source code is very simple [10]. 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 *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 *wait*(t,k) that waits while $t < k$. The *reset*(i,k) command programs an interrupt i to arise after k seconds. The *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 are removed since they do not describe program instructions but merely non-functional properties. Concretization uses the following rules:

- couples of transitions of the form

$$(q_1 \xrightarrow{.,\mathbb{B}(c),.} q',\ \ q' \xrightarrow{.,\mathbb{E}(c),.} q_2)$$

correspond to a command c and are translated into the instruction $l_{q1} : c \leadsto l_{q2}$;

- tuples of transitions of the form

$$\begin{array}{ll}(& q_1 \xrightarrow{.,\mathbb{B}(g),.} q',\ q' \xrightarrow{.,\mathbb{E}(g),.} q_2, \\ & q_1 \xrightarrow{.,\mathbb{B}(\neg g),.} q'',\ q'' \xrightarrow{.,\mathbb{E}(\neg g),.} q_3)\end{array}$$

correspond to a guard g and are translated into the instruction $l_{q1} : g \leadsto l_{q2} \ ; \ l_{q3}$;

- couples of transitions of the form

$$(q_1 \xrightarrow{g \wedge G,\mathbb{B}(a),.} q',\ q_1 \xrightarrow{\neg g \wedge G,\mathbb{B}(a),.} q'')$$

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

$$(q_1' \xrightarrow{G,\mathbb{B}(a),.} q',\ q_1'' \xrightarrow{G,\mathbb{B}(a),.} q'')$$

- a loop $q \xrightarrow{t<k,\mathbb{B}(\mathit{wait}),.} q' \xrightarrow{t \geq k,\mathbb{E}(\mathit{wait}),.} q$ 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.*, $l_{q'}$);
- interrupts 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 9 shows the source code of service S1 obtained after the concretization of the automaton of figure 8. 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 its treatment, the resource M1 is released (`M1.free()`) and the interrupt is canceled (*cancel* (i)).

$$\texttt{S1} = \left\{ \begin{array}{llll} l_0 & : & \texttt{getUser}() & \leadsto \ l_1 \\ l_1 & : & \texttt{M1.alloc}() & \leadsto \ l_1' \\ l_1' & : & \mathit{reset}(i,25) & \leadsto \ l_2 \\ l_2 & : & \texttt{M2.alloc}() & \leadsto \ l_3 \\ l_3 & : & \texttt{S1Comput}() & \leadsto \ l_4 \\ l_4 & : & \texttt{M2.free}() & \leadsto \ l_5 \\ l_5 & : & \texttt{M1.free}() & \leadsto \ l_6' \\ l_6' & : & \mathit{cancel}(i) & \leadsto \ l_6 \\ l_6 & : & \texttt{endUser}() & \leadsto \ l_0 \end{array} \right\}$$

Figure 9. Code of service S1 after weaving

8. 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 services. 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 [13] 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 easy and realistic. The representation 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 [6]).

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 product.

- in [6], 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 [10], 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. Several AOP related approaches also rely on automata. Let us mention [18] which uses automata to enforce safety properties and [24] and [1] which investigate AOP for parallel languages. Their respective goals and techniques are quite different from ours; in particular, none of them consider timed properties and automata.

Yu and Gligor [25] present a method to verify that a resource allocator remains available. Our framework can be seen as a generalization of that work to bounded time policies. Further, aspects allow a better separation of concerns and, above all, an automatic instrumentation of programs using weaving. Millen [19] proposes a model based on a global monitor for availability relying on a *Trusted Computing Base*. We have shown that local policies can suffice to ensure availability properties. Local policies are easier to design and to implement efficiently. Cuppens and Saurel [7] presents a logic framework to express and verify availability policies. Their model is suitable to verify policies *a posteriori* but not to enforce them. J-Seal2 [5] proposes a simple and global mechanism to ensure availability of processors and memory. They describe the implementation 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 [20] abstracts a TCP server into a timed automaton and use UPPAAL to verify its ability to survive denial-of-service attacks. They do not consider the enforcement availability properties but we could reuse their timing analysis to abstract our services.

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

Acknowledgments

This work has been supported by the ACI DISPO project. We thank the anonymous reviewers for their corrections and suggestions.

References

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

[2] R. Alur. Timed automata. In *Computer Aided Verification*, pages 8–22, 1999.

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

[4] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Concurrency and Petri Nets*, LNCS vol. 3098. Springer–Verlag, 2004.

[5] W. Binder, J. G. Hulaas, and 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*, pages 139–155. ACM Press, 2001.

[6] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Symposium on Principles of Programming Languages (POPL'00)*, pages 54–66, 2000.

[7] F. Cuppens and 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.

[8] R. Douence, P. Fradet, and 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.

[9] R. Douence, P. Fradet, and M. Südholt. Trace-based aspects. In M. Aksit, S. Clarke, T. Elrad, and R. Filman, editors, *Aspect-Oriented Software Development*, pages 201–217. Addison-Wesley, 2004.

[10] P. Fradet and S. Hong Tuan Ha. Network fusion. In *Prog. Lang. and Syst.: Second Asian Symposium, (APLAS'04)*, LNCS vol. 3302, 2004.

[11] P. Fradet and 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)*, Sept. 2005.

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

[13] S. Hong Tuan Ha. *Programmation par aspects et tissage de propriétés. Application à l'ordonnancement et à la disponibilité.* PhD thesis, Rennes University, Jan. 2007.

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

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

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

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

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

[19] J. K. Millen. A resource allocation model for denial of service protection. *Journal of Computer Security*, 2(2), 1994.

[20] V. K. Nandivada and J. Palsberg. Timing analysis of tcp servers for surviving denial-of-service attacks. In *IEEE Real-Time and Embedded Technology and Applications Symp.*, pages 541–549, 2005.

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

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

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

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

[25] C.-F. Yu and V. D. Gligor. A specifi cation and verifi cation method for preventing denial of service. *IEEE Trans. Soft. Eng.*, 16(6), 1990.