On the Semantics of the GenoM3 Framework

Mohammed Foughali, Silvano Dal Zilio, Félix Ingrand

On the Semantics of the GenoM3 Framework

Mohammed Foughali,
Silvano Dal Zilio, and Félix Ingrand

CNRS, LAAS, 7 avenue du Colonel Roche, F-31400 Toulouse, France
Université de Toulouse, LAAS, F-31400 Toulouse, France

1 Introduction

In the current practice, robotic software trustworthiness relies on testing campaigns, best coding practices, and the choice of sound architecture principles. While such methods are helpful, they unfortunately do not provide guarantees on crucial *properties* such as *schedulability* of tasks, *leads to* and *bounded response*. Such properties often reflect the requirements on the safety and predictability of the system. For instance, the bounded response is crucial in self-driving cars, where we want to know that for all possible execution scenarios, the maximum time difference between requesting a brake action and the actual braking is small enough so the vehicle stops before colliding with an obstacle (which might be a pedestrian). Verifying this type of properties is thus necessary to obtain a high level of trust in the robotic software, yet the routinely employed methods fall short of giving the desired answers. For instance, scenario-based testing is widely used in robotics but cannot, due to its non-exhaustive nature, verify a property with a known level of certainty. Consequently, the reliability of robotic software does not rise to the level found in many regulated domains, such as the aeronautic or nuclear industries, where *formal methods* are used to check the most vital parts of systems [33].

The question that arises is why formal methods are not systematically employed to verify robotic software? There exist many reasons, emerging from the specificities of the robotics domain, such as the unstructured nature of environments, compared to other domains like aeronautics. But, more generally, to answer this question , we need to differentiate the levels of robotic software [2, 32], mostly viewed as *functional* (tightly coupled with sensors and actuators) and *decisional* (in charge of *deliberative* functions [19]). In contrast to most of the decisional ones, functional components are specified within informal frameworks such as ROS [27] and Orocos [10]. Thus, in order to apply formal methods to the functional level, we need first to *formalize* its specifications. The formalization is hard and error-prone and is consequently the most time-consuming and error-prone step in verification [26]. Also, the transformation chain, from robotic specifications to formal models, is not automatic (it needs to be re-done from scratch for each new application). Additionally, there is a large number of existing formalisms/tools that can be employed in modeling and verification. The mutual advantages and drawbacks of such formalisms/tools depend on the applications/properties to verify and cannot thus be known beforehand [11, 31]. In practice, the high cost of formalization and the absence of automation limit the choice to only one formalism/tool, which makes it impossible to know whether verification might be improved with other formalisms/tools. Moreover, the complexity of the functional level (e.g. number of *components*, timing constraints, communication mechanisms) leads frequently to

scalability issues. Overall, there is a visible gap between the robotic and formal methods communities. On one hand, robotic programmers have neither the knowledge nor the time to invest in applying formal methods to their applications. On the other hand, formal methods specialists are often far from dealing with systems as complex as the robotic ones.

Proposition The goal of this document is to add to the efforts toward the long-sought objective of secure and safe robots with predictable and *a priori* known behavior. For this, we give operational semantics to GenoM3, a robotic framework, in terms of timed transition systems TTS (Sect. 4.1). Then, a mathematically proven translation to timed automata extended with urgencies and data DUTA (Sect. 4.2) is derived from such semantics. Thus, we provide a mapping from functional components to verifiable models. Since TTS and DUTA are at the heart of a large corpus of formal verification languages and tools (such as UPPAAL [5], Fiacre [7], and RT-BIP [4]), the semantics and its translation allow a correct mapping between GenoM3 and such languages/tools. This connection can then be automatized thanks to GenoM3 templates [23].

1.1 Related work

To cope with the problem of lack of semantics at the functional level in robotics, some works propose alternative languages to specify robotic applications, such as the synchronous language ESTEREL [9] (see [6] for synchronous languages). ESTEREL is used many works like [29, 30, 21], where at times the robotic specifications are directly hard-coded in this language. Using formal languages directly to encode robotic specifications is not suitable for the robotic community, rather familiar with robotic frameworks.

RoboChart [25] is used in several verification efforts such as [24]. RoboChart models are automatically translated into Communicating Sequential Processes (CSP) [28] in order to verify behavioral and timed properties using the FDR model checker [14]. RoboChart is a UML-like notation with formal semantics, but is not a robotic framework per se (its models are not executable on robotic platforms). That is, one still needs to have applications already written using a robotic framework to translate them into RoboChart. So far, there is no proven translation approach linking a robotic framework to RoboChart.

In the literature, we find a number of attempts to formalize robotic frameworks, especially the most popular among them, ROS. The authors of [16] use the PRISM probabilistic model checker [22] to verify bounded response properties on ROS applications. They attempt to formalize ROS graphs, but no operational semantics is given which makes the formalization both manual and ad-hoc. Another attempt to formalize ROS components is developed in [15] where UPPAAL is used to verify buffer-related properties (no overflow). ROS components are not formalized and only the message passing part (publisher/subscriber) is modeled.

Another way to go around the lack of semantics at the functional level is to rely on runtime verification. BIP [4], a modeling and verification framework based on automata, is used in the verification effort presented in [1]. The functional components, written in GenoM (version 2), of an outdoor robot with two navigation modes, are modeled in BIP. Safety constraints, such as “the robot must never communicate and move at

the same time" are automatically translated from logical formulae into BIP then added to the model. The latter is run within the BIP-Engine on DALA, an iRobot ATRV, and the constraints are consequently enforced at runtime. It is not possible to verify the soundness of the translation from GenoM to BIP due to the absence of operational semantics of the former.

1.2 Contributions

Contribution 1 As shown so far, formalization is particularly difficult and error prone at the functional level. This is due, mainly, to the fact that component-based frameworks in robotics are not formal, which makes their formalization output questionable in terms of their correctness vis-à-vis its robotic input. To solve this problem, we propose formal semantics for GenoM3, our chosen robotic component-based framework for this document. Components have thus formal definitions and their operational semantics is developed formally (Sect. 5) in TTS (Sect. 4.1). So far, formalization of robotic frameworks at the functional level range from shy to inexistent 1.1.

Contribution 2 The semantics is translated into DUTA and the translation is proven sound. TTS (respect. DUTA) are the underlying formalism for formal languages such as Fiacre (respect. for major verification frameworks/tools such as UPPAAL and BIP). By having the semantics in TTS and its counterpart sound translation in DUTA, mapping to tools based on these formalisms is already halfway done. This opens the door wide for automatic translations from GenoM3 to several formal frameworks and exploiting their respective advantages.

1.3 Outline

The remainder of this document is organized as follows. First, we give an informal description of GenoM3, from requirements to behavior. Then, we introduce in Sect. 4 TTS, and motivate choosing them to formalize GenoM3. Afterwards, we define within the same section timed automata and their different extensions with urgencies and data variables. In Sect. 5, we present the core of the work of semanticizing a lightweight version of GenoM3. A translation from such semantics to an extended version of timed automata is presented then in Sect. 6, together with a proof of soundness of the translation. Finally, we conclude with presenting the benefits this work brings to the broad domain of formal verification of robotic systems.

2 GenoM3

3 Overview

GenoM3 [23] is a tool to specify and implement robotic functional components. The LAAS architecture [20] proposes a modular approach where each functional component acts as a "server" in charge of a given functionality. The latter may range from simple low-level driver control (e.g. the velocity control of the propellers of a drone, camera) to more integrated computations (e.g. Simultaneous Localization And Mapping (SLAM), Potential-Field navigation, Rapidly-exploring Random Tree (RRT) motion planning).

3.1 Requirements

We consider that a typical component is a *program* which needs to manage the following aspects:

Inputs and Outputs : a component interacts with *external clients* and other components. For the former, the control flow, it must handle *requests* from client(s) and send back *reports* to the client which issued the request, to act on the result. For the latter, the data flow, it must provide a mechanism to share data with other components.

Algorithms : the core algorithms needed to implement the functionality the component is in charge of must be appropriately organized within *e.g. services* (Sect.3.2) to fulfill the clients requests. A component may have just one service to provide, but most of the time, there are a number of such services associated to the considered robotic functionality. The way algorithms are specified and organized in a component is a tradeoff. One can let the programmer organize their code with no design requirements or provide structure guidelines that must be followed. The latter case enables formalization, given that the set of guidelines is well known (Sect. 5). These organization rules must remain simple and easily understandable for robotic programmers.

Data sharing : the various algorithms, possibly concurrent, running in the component, may have to share data that represent the internal state of the component. These data need to be handled correctly respecting *e.g.* mutual exclusion conditions.

3.2 Organization

To achieve such requirements of a functional component, we propose to organize each one along the structure shown in Fig. 1. Specifying components in GenoM3 is the programmer's design choice. Thus, there are a number of considerations, depending on various factors such as hardware constraints and algorithms complexity, that they have to take into account. Here, we describe in more details the different elements of a GenoM3 component, and how they are specified, in a generic manner. That is, the description given here is not specific to any implementation or component.

To ease the comprehension of the different elements, a support example is given in listing 1.1. It shows the *dotgen* (extension *.gen*) specification of a simple GenoM3 component called DEMO developed for illustration purposes. DEMO is a simple one-dimensional motion component of a mobile robot. Its main functionalities are to move the mobile for a relative distance within the interval $[-1, 1]$, to monitor its position, to read its speed, and to change it. The elements in charge of these operations are given within the description of the component constituents hereafter.

Control Task : A component always has a *control task* that manages the control flow by processing *requests* and sending *reports* (from/to external clients). The control task must be highly reactive and is only assigned quick computations. It also manages interruption and activation of longer computations (see more in Sect. 3.3). The control task is implicitly comprised within a component and the user does not specify it, hence its absence from listing 1.1.

Execution Task(s) : Aside from the *control task*, one may need one or more *execution tasks*, aperiodic or periodic, in charge of longer computations. The component DEMO has one execution task called motion (periodic at 400 ms, lines 17-19).

Services : The core algorithms needed to implement the functionality of the component are encapsulated within *services*. Each *service* is associated to a *request* (with

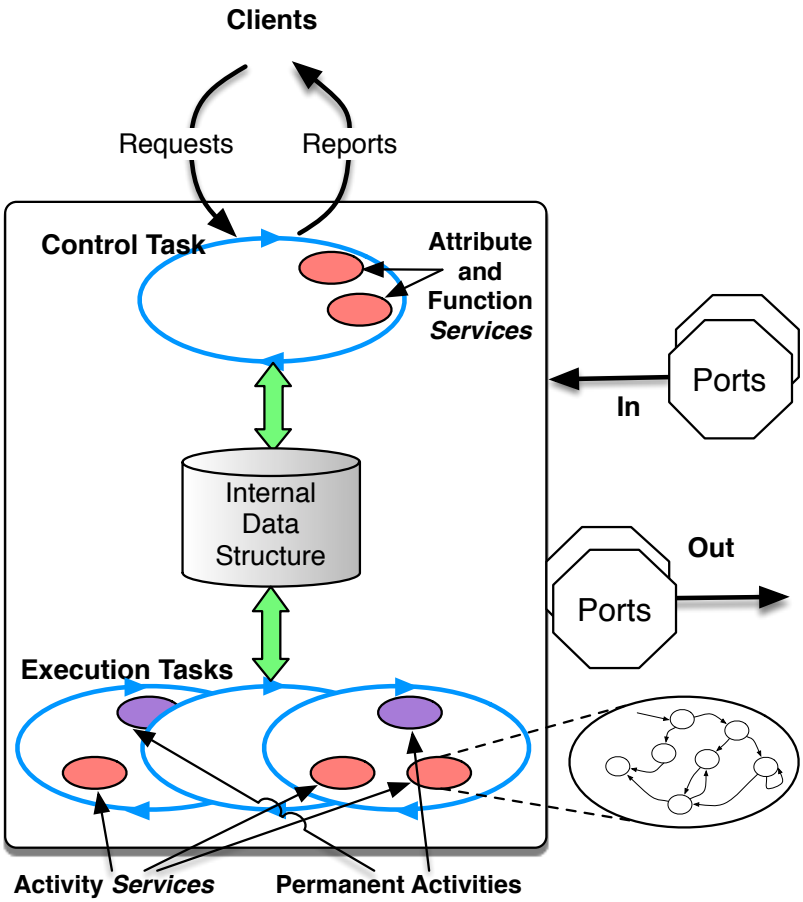


Fig. 1: A generic GenoM3 component.

the same name). One may also define a *permanent service* (running without being requested) attached to each *execution task*. In the DEMO component, services are *MoveDistance* (move the mobile for a relative distance within $[-1, 1]$, lines 34-45), *Monitor* (monitor the position, lines 46-54), *GetSpeed* (get the current speed, lines 26-27), *SetSpeed* (change the current speed, lines 22-25), *Finish* (stop moving, lines 29-32), and the permanent service of motion (initialization, line 19).

IDS : A local *internal data structure* is provided for all the services to share parameters, computed values and variables of the component. It is appropriately accessed (i.e. with proper locking) by the services when they need to read or write one or more of its fields (lines 4-7). For instance, the arguments of *GetSpeed* specify that it reads the current speed from the IDS (line 26).

Ports : They specify the shared data and the access direction to them (read "in" or write "out"), the component needs/produces from/for other components. The component DEMO provides one port that it writes (*out* mode, line 10).

Exceptions : One may specify *exceptions*, which can be returned by services to *report* on execution errors (lines 13-14).

3.3 Behavior

We go in more details and see how these different elements interact and how the component internally runs.

Codels Code elements, or codels, are small chunks of C or C++ code (*e.g.* the codel `StopMotion` in line 31 matches a C function whose body is defined in a separate file). When defined within *activities*, codels are associated with *states* in a finite-state machine (see *activities* and *FSM* below). For instance, the codel `mdGotoPosition` (line 39) is associated with the state `exec` (more details below).

Services Services hold the specifications of the algorithms handled by the component. Services can take arguments (*e.g. SetSpeed* takes a *SpeedRef*, line 22), and return values (*e.g. GetSpeed* outputs a *SpeedRef*, line 26). A service may have a `validate` codel (*e.g. Monitor*, line 49). When the control task receives a service request, it runs the service `validate` codel, if any, to check whether the service arguments are valid (it reports an error to the client that requested it if they are not). A service may also specify activities (see below) it interrupts (*e.g. Finish* interrupts *MoveDistance*, line 32). If a service S interrupts an activity S', we say that S' is *incompatible* with S. Aside from control services (see below), a service may not run unless all the services it interrupts are terminated. A service that is ready to run is called an *activated* service. There are two types of services:

Control Services, are only for quick computations which should not delay the control task (that executes them). A control service may be an *attribute* (setter or getter of fields of the IDS, *e.g. GetSpeed*), or a *function* (for quick and simple computations, *e.g. Finish*).

Activities, are executed by the execution task specified in their declaration (*e.g.* line 44, the activity *MoveDistance* is executed by the task motion). Activities are *finite-state machines*, each state associated with a codel (see codels above).

FSM define the behavior of the activity through states, codels and *transitions*. A codel specifies the state it is associated to and the C or C++ function it will call, with the arguments (taken from the activity arguments, the IDS and the ports of the component) they need for their execution (*e.g.* `mdGotoPosition` is associated with the state `exec` of *MoveDistance*, it reads the fields *speedRef* and *posRef* of the IDS and writes the field *state* of the IDS and the port **Mobile**, line 39). A codel specifies also the possible *transitions* subsequent to its execution (*e.g.* the execution `mdStartEngine`, associated with the state `start` of *MoveDistance*, returns the state `exec` or the state `ether`, lines 37-38). The non-determinism is resolved at runtime when executing the codel, which returns upon completion the next state to transit to. Taking a transition labeled *pause* stops the execution of the activity until the next cycle of its execution task (see execution tasks below), the activity is thus *paused* (*e.g.* if the execution of `monitor` in activity *Monitor* returns `pause::start`, line 50, *Monitor* is paused at state `start` until the next cycle of the task motion). Each codel may (optionally) specify a WCET, namely its worst case execution time on a given platform (*e.g.* `mdStopEngine`, associated with the

```
/* ---- component declaration ---- */
component demo {
/* ---- Data structures and IDS ---- */
ids {
       demo::state state;  /* Current state */
       demo::speed speedRef;  /* Speed reference */
       double posRef;};

/* ports declaration: direction type name */
port out demo::state Mobile;

/* exception declaration */
exception TOO_FAR_AWAY {double overshoot;};
exception INVALID_SPEED;

/* execution tasks declaration */
task motion {
       period 400 ms;
       codel <start> InitDemoSDI(out ::ids, port out Mobile) yield ether;};
/* services declaration */
/* atributes */
attribute SetSpeed(in speedRef :"Mobile speed") {
       doc  "To change speed";
       validate controlSpeed (local in speedRef);
       throw INVALID_SPEED;};
attribute GetSpeed(out speedRef = :"Mobile speed") {
       doc  "To get current speed value";};
/* functions */
function Finish() {
       doc  "Stops motion and interrupts all motion requests";
       codel StopMotion();
       interrupts MoveDistance;};
/* activities */
activity MoveDistance(in double distRef :"Distance in m") {
       doc  "Move of the given distance";
       validate controlDistance(in distRef, in state.position);
       codel <start> mdStartEngine(in distRef, in state.position, out
           posRef)
              yield exec, ether;
       codel <exec> mdGotoPosition(in speedRef, in posRef, out state, port
           out Mobile)
              yield exec, end;
       codel <end> mdStopEngine() yield ether wcet 1 ms;
       codel <stop> mdStopEngine() yield ether;
       interrupts MoveDistance;
       task motion;
       throw TOO_FAR_AWAY;};
activity Monitor (in double monitor = 0 :"Monitored absolute position in m",
              out double position) {
       doc  "Monitor the passage on the given position";
       validate controlPosition (in monitor);
       codel <start> monitor(in monitor, in ::ids) yield pause::start, end
           wcet 2 ms;
       codel <end> monitorStop(in ::ids, out position) yield ether;
       codel <stop> monitorStop(in ::ids, out position) yield ether;
       task motion;
       throw TOO_FAR_AWAY;};
};
```

Listing 1.1: Excerpt from the GenoM3 specification of the DEMO component.

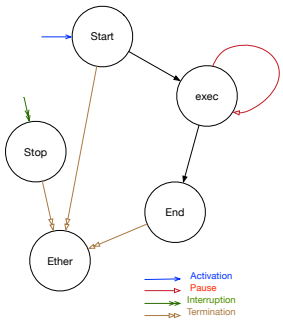


Fig. 2: FSM of *MoveDistance* (lines 34-45 of listing 1.1).

state `end` of *MoveDistance*, has a WCET of 1 ms, line 41). Any activity FSM has the states `start` (entry point) and `ether` (end point). When the latter is reached, the activity is *terminated* and reported to the client. The state `stop`, if exists, is associated with the codel to execute when the activity is interrupted (*e.g.* line 51). If an activity with no `stop` codel is interrupted, it transits directly to `ether`. Fig. 2 is a visual illustration of the FSM behavior of activity *MoveDistance* (WCETs are omitted).

The organisation of activities along FSMs may be seen wrongfully as an unnecessary burden for programmers. Indeed, nothing prevents the programmer from having one `start` codel that does it all. Yet, breaking code along an FSM brings a number of advantages as it *e.g.* improves code execution interleaving and provides a finer model of data sharing and code interlocking (several short computations using each a fragment of resources brings a better concurrency level and allows shorter task periods than a single long computation that uses all resources). Furthermore, FSMs are amenable to translation into formal languages (Sect. 5).

Control task The control task has a cyclic behavior that consists in managing the requests and reports of the component, executing control services and activating and interrupting activities. When a service request is received, the control task instructs the execution tasks to interrupt the activities incompatible with the requested service. If the latter is a control service (attribute or function), the control task executes it immediately. If it is an activity, it is put on hold until all the incompatible instances are correctly interrupted and terminated. It is then activated, (may be run by the execution task in charge of it). Upon completion of any service, the control task sends a report to the corresponding client (service ended nominally, interrupted, or failed by throwing an exception).

Execution tasks Execution tasks are cyclic tasks that can be periodic or aperiodic (*e.g.* the period of motion is 400 ms, line 18). With each cycle (triggered by a period signal or event occurrence), the execution task runs, sequentially, its permanent activity (if any) and all the instances of the activities it is in charge of, previously activated by the control task. The execution of an activated instance ends when the instance is paused or terminated. In the former case, the instance will be resumed at the next cycle.

Internal Data Structure The IDS stores data that represent the internal state of the component, shared among tasks and services. For instance, the IDS of DEMO (lines 4-7) stores the current position and current speed (in the field *state*) , the speed reference and the position reference of the mobile. Access to the IDS is mutually exclusive. One can see that the proper specifications (enforced by GenoM3) of the codel arguments allows for a fine grain locking of the IDS and thus a high level of concurrency (only the needed field(s) by a codel are locked when it executes and simultaneous readings are allowed).

Ports Data flow between components is made through ports (line 10). As seen above, ports usage (in or out) is also declared in codels arguments (*e.g.* line 39). Consequently, over a large set of components composing a robotic functional layer, we have a clear model of which codels use a particular port. Ports allow functional-layer components to exchange data. In a sensor-based navigation application, for instance, the collaboration of several components is indispensable (*e.g.* the information collected by a component from a sensor is a necessary input for another component handling a controller).

Clients GenoM3 components are usually unable to evolve unless controlled by external clients. Indeed, apart from permanent activities, services need to be requested in order to be served by the component. For instance, after being implemented and run, the component DEMO, whose specification is shown in listing 1.1, does not execute any service. Indeed, the control task, in charge of the component, needs to receive requests in order to run control services and activate activities. For this, clients, which are external entities to the component, send the requests for the services they want to run, together with the arguments, if any. For example, the following line in a *Tcl* client requests the activity *MoveDistance* with the argument 0.5, that is requests moving the mobile for 0.5m:

```
demo::MoveDistance(0.5)
```

4 Formalisms

4.1 Timed Transition Systems (TTS)

The chosen formalism is a variation of the Timed Transition Systems (TTS) presented in [17]. There are several arguments that justify this choice and that will be given later in this document (Sect. 4.1.7).

One difference between our definition of TTS and the one proposed in [17] is that we consider a dense-time model (durations and time constraints have values in $\mathbb{R}_{\geq 0}$ with interval bounds in $\mathbb{Q}_{\geq 0} \cup \infty$) whereas the original presentation relies on a discrete-time model (durations have values in $\mathbb{N}$). We also accept more general timing constraints, using time intervals with possibly left-open and right-open bounds.

4.1.1 Notations We start by defining some notations that will be useful in the remainder of this document. We use $\mathbb{I}$ to denote the set of well-formed (time) intervals over positive reals, with rational lower bounds and rational or infinite upper bounds. An element i of $\mathbb{I}$ can be of one of four types: (where $a \in \mathbb{Q}_{\geq 0}$ and b can be either a rational number or the infinity symbol, ∞, meaning an infinite bound).

– $[a, b]$ (with $a \leqslant b$),

– $]a, b]$ (with $a < b$),
– $[a, b[$ (with $a < b$),
– $]a, b[$ (with $a < b$).

We say that $\downarrow i = a$ is the lower bound of the interval i and $\uparrow i = b$ is its upper bound.

In the following, we use the notation $\sqsubset a, b \sqsupset$ for time intervals, where $\sqsubset$ (respect. $\sqsupset$) denotes whether i is open or closed at its left (respect. right) bound. Therefore we have $\sqsubset = [$ for a closed interval on the left and $\sqsupset = [$ for an open (strict) interval on the right. Likewise we use ']' for an open interval on the left and closed interval on the right. By an abuse of notation, we also conflate open/close interval symbols with comparison operators between reals. We say that $\sqsubset$ is the strict comparison operator $<$ when i is left-open ($\sqsubset =]$) and that $\sqsubset$ is the operator $\leqslant$ when i is left-closed ($\sqsubset = [$). Likewise, we say that $\sqsupset$ is the operator $\leqslant$ when i is right-closed and $<$ otherwise. With this choice of notation, an interval $i = \sqsubset a, b \sqsupset$ is exactly the set of real values $x \in \mathbb{R}_{\geq 0}$ such that $a \sqsubset x$ and $x \sqsupset b$.

For any date δ in $\mathbb{Q}_{\geq 0}$ and interval $i \in \mathbb{I}$, we denote $i - \delta$ the time interval obtained by shifting i (to the left) by an amount of time δ. The operation is defined only if $\delta < \uparrow i$ (or if $\delta \leq \uparrow i$ and i is right-closed), which we call the *upper bound condition*. We consider four different cases depending on the "shape" of interval i. Assume $a' = \max(0, a - \delta)$:

– if $i = [a, b]$ and $\delta \leqslant b$ then $i - \delta = [a', b - \delta]$,
– if $i =]a, b]$ and $\delta \leqslant b$ then $i - \delta =]a - \delta, b - \delta]$ if $\delta \leqslant a$ and $[0, b - \delta]$ otherwise,
– if $i = [a, b[$ and $\delta < b$ then $i - \delta = [a', b - \delta[$,
– if $i =]a, b[$ and $\delta < b$ then $i - \delta =]a - \delta, b - \delta[$ if $\delta \leqslant a$ and $[0, b - \delta[$ otherwise (With the convention that $\infty - \delta = \infty$).

4.1.2 Syntax of TTS A Timed Transition System TTS is a tuple $\langle U, S, s_0, \tau, I \rangle$ where:

– U is a finite set of variables. Each variable is implicitly typed. We use $\mathrm{dom}(u)$ to denote the domain of variable u in U;
– S is a set of states. Each state of S is an interpretation of variables in U, that is a mapping from variables $u \in U$ to values in $\mathrm{dom}(u)$;
– s_0 is the initial state ($s_0 \in S$) that maps each variable to its initial value;
– τ is a set of transitions. Each transition $t \in \tau$ defines a (possibly empty) set of *successor states*, denoted $t(s)$, for every state $s \in S$;
– $I : \tau \mapsto \mathbb{I}$ maps each transition $t \in \tau$ to a *static (time) interval* $I(t) \in \mathbb{I}$.

A transition $t \in \tau$ is said *enabled* at s if and only if s is a *source state* of t, that is $t(s) \neq \varnothing$. We denote $\mathcal{E}(s)$ the set of transitions enabled at s.

We require the set $t(s)$ to have cardinality at most one for any t in τ and any s in S. This allows us to simplify the presentation of the semantics (especially when defining the notion of persistent transitions later in this section) without loosing any expressiveness (a state s may still have many successors over transitions with different "names").

4.1.3 Semantics of TTS In a TTS $\langle U, S, s_0, \tau, I \rangle$, each (enabled) transition is associated with a timing constraint, that is an interval $I(t) = \sqsubset a, b \sqsupset \in \mathbb{I}$. The semantics of time depends on the dates at which the transition can be *taken*. Informally, if s is the current state since the date Δ and if transition t is enabled, then we can *take* t starting at a date d s.t. $\Delta + \downarrow I(t) \sqsubset d$ and no later than a date $d' \sqsupset \Delta + \uparrow I(t)$, unless t is disabled in between by taking another transition.

The semantics of a TTS is therefore given over pairs (s, ϕ) where $s \in S$ is a state and $\phi : \tau \to \mathbb{I}$ is a mapping from transitions to time intervals. Intuitively, if t is enabled at s, then $\phi(t)$ contains the dates at which t can be possibly taken in the future. Hence, a transition t can be taken (immediately) only when 0 is in $\phi(t)$. Likewise, a transition t cannot remain enabled for more than its timespan, that is the value $\uparrow\phi(t)$.

We use $\phi \dot{-} \delta$ for the partial function that associates, at a state s, each transition $t \in \mathcal{E}(s)$ to the value $\phi(t) - \delta$ (the interval $\phi(t)$ shifted by δ). This function is useful to model the effect of time progress on the enabled transitions in a TTS. (Note that $\phi \dot{-} \delta$ is defined only when $\phi(t) - \delta$ is defined for all $t \in \mathcal{E}(s)$, that is δ satisfies the upper bound condition for all $\phi(t)$).

Let t be enabled at s with $t(s) = \{s'\}$. We say that a transition k is *persistent* (with $k \neq t$) if it is also enabled at s'. The transitions that are enabled at s' and not at s are called *newly enabled*. We define the predicates $pers(s, t)$ and $nenabl(s, t)$ that describe, respectively, the sets of persistent and newly enabled transitions after t is taken from s. We see that if t is still enabled in s' then it is necessarily newly enabled.

$$\begin{aligned} pers(s,t) &= \{k \in \tau \mid k \in \mathcal{E}(s) \wedge k \in (\mathcal{E}(s') \setminus \{t\}) \wedge t(s) = \{s'\}\} \\ nenabl(s,t) &= \{k \in \tau \mid k \notin (\mathcal{E}(s) \setminus \{t\}) \wedge k \in \mathcal{E}(s') \wedge t(s) = \{s'\}\} \end{aligned}$$

With all these notations, we can define the semantics of a TTS as a Kripke structure (a rooted, *state graph*) such that:

- states in the graph are pairs (s, ϕ) where $s \in S$ is a state and ϕ is a mapping from $t \in \mathcal{E}(s)$ to $\mathbb{I}$,
- the initial state is (s_0, ϕ_0) where ϕ_0 is such that $\phi_0(t) = I(t)$ for each transition $t \in \mathcal{E}(s_0)$ (all transitions possible from s_0 are newly enabled),
- *discrete transitions*: from every reachable state (s, ϕ) and every transition $t \in \mathcal{E}(s)$, we have a (discrete) transition $(s, \phi) \xrightarrow{t} (s', \phi')$ when $0 \in \phi(t)$ and $t(s) = \{s'\}$. In this case ϕ' is the unique mapping such that: $\phi'(k) = \phi(k)$ for all transitions $k \in pers(s, t)$ and $\phi'(k) = I(k)$ otherwise,
- *continuous transitions*: for every delay $\delta \in \mathbb{Q}_{\geq 0}$ such that $\phi \dot{-} \delta$ is defined, we have a (continuous) transition $(s, \phi) \xrightarrow{\delta} (s, \phi \dot{-} \delta)$.

From this definition, we see that time progress does not change the set of enabled transitions; but it may change the set of transitions that may be taken immediately (the set of transitions such that $0 \in \phi(t)$). We can also see that the state graph of a TTS is generally infinite. Indeed, in most cases, we can choose between an infinite number of continuous transitions. This is, for instance, the case when there are no transitions in τ enabled at s (in which case we can let time elapse by an unbounded amount).

We give, In Table 1, an alternate definition of the reduction relation using notations borrowed from structural operational semantics, where the relation $\to$ is defined by a

$$\text{(discrete)}\ \frac{t \in \mathcal{E}(s) \qquad 0 \in \phi(t) \qquad t(s) = \{s'\} \qquad \forall k \in \mathcal{E}(s') : \phi'(k) = \phi(k) \text{ if } k \in pers(s,t) \text{ and } I(k) \text{ otherwise}}{(s,\phi) \xrightarrow{t} (s',\phi')}$$

$$\text{(continuous)}\ \frac{\delta \in \mathbb{Q}_{\geq 0} \qquad \phi \dot{-} \delta \text{ defined}}{(s,\phi) \xrightarrow{\delta} (s, \phi \dot{-} \delta)}$$

Table 1: Operational Semantics of TTS.

set of inference rules. We use this notation later in order to simplify the presentation of the semantics of GenMSM and its translation to DUTA.

TTS in this document follow a *strong time semantics*, meaning that we must always take an enabled transition from a state (s, ϕ) if there is no delay $\delta > 0$ such that $\phi \dot{-} \delta$ is defined (that is when time cannot elapse). Since a transition cannot become disabled from a continuous transitions, it follows that a TTS cannot have a "timelock", that is a situation in which a system is blocked because all timing constraints are indefinitely false.

4.1.4 Timed Transition Diagrams We define a graphical notation for TTS (called Time Transition Diagrams, or TTD for short) as well as a composition operation between TTDs that is also inspired from the work in [17]. Basically, we can see every TTD as a component and composition as a way to build more complex systems through the synchronization and interactions of simpler systems. In this approach, the composition of multiple TTDs (viewed as *components*) results in a TTS (viewed as the *system*).

A timed transition diagram (TTD) P is a finite directed graph, operating on a finite set of variables, Y. V is the set of vertices and E the set of edges in the graph. The vertex $v_0 \in V$ denotes the unique initial vertex of P. Each *edge* $e \in E$ is associated with: an interval $I(e)$; a guard g_e; and operations op_e. If an edge e connects vertex v to vertex v', then we write by an abuse of notation, interchangeably, $e \in E$ or $v \xrightarrow{e} v' \in E$ to denote such edge. In the remainder of this document, guards that are always true, operations with no effect on variables and $[0, \infty[$ intervals will not be represented.

As with TTS, we say that a state s of a TTD is a mapping from variables to values. We consider a distinguished variable, or *control vertex*, denoted π, whose value gives the "current vertex" of the TTD. Hence $\mathrm{dom}(\pi) = V$ and the initial value of π is v_0.

Informally, a guard is a boolean expression over Y that defines when an edge is enabled, whereas operations are instructions that can modify the values stored in these variables (a sequence of operations is processed atomically). We use the expression $g(s)$ to denote the "truth" value of a guard g at the state s. likewise, we use the notation $s'_{|Y} = op(s_{|Y})$ when the results of op on Y from s agree with the interpretation of Y at s'. In particular, $s'(y) = s(y)$ for each y in Y that is not affected by op.

We show in Fig. 3 a simple generic TTD example with two vertices, v_0 and v_1, and one edge $v_0 \xrightarrow{e} v_1$. The initial vertex, in this case v_0, is denoted with an incoming edge without source vertex.

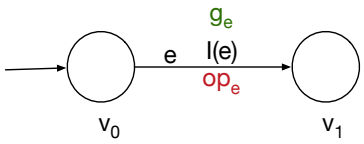


Fig. 3: A generic TTD example

Given a TTD P, we can associate its meaning, $[\![P]\!]$, that is a TTS that corresponds to P. The meaning of P is the TTS $\langle U = Y \cup \{\pi\}, S, s_0, \tau, I\rangle$ such that:

- S is the set of states, where each state is an interpretation of π and each variable in Y.
- the initial state s_0 is the mapping associating π to v_0 (initially the control vertex is at v_0) and all the variables in Y to their initial value,
- τ is the set of transitions resulting from mapping each edge e in E to a transition t_e as follows. If e connects vertice v_a to v_b, then $s' \in t_e(s)$ iff
  $s(\pi) = v_a$ and $s'(\pi) = v_b$; and
  $g_e(s)$ is true; and
  $s'_{|Y} = op_e(s_{|Y})$.
- The function I maps every transition t_e to the interval $I(e)$.

4.1.5 Composition of TTDs The parallel composition of a finite number of TTDs, $P_1, \ldots, P_n$, over a set of shared variables, U_s, results in a TTS denoted:

$$\{\Theta\}[\,\|_{i\in 1..n}\, P_i\,]$$

A TTS also defines an initial valuation, Θ, that gives the initial assignation of variables in U_s to values.

Edge (identifiers) of different components are always distinct: if e is an edge in P_i then it cannot be an edge in P_j with $i \neq j$. Also, each component (TTD) P_i can have access to a set of local variables, denoted U_i, besides the variables in U_s ($U_i \cap U_s = \varnothing$ and $U_i \cap U_j = \varnothing$ for all indexes $i, j \in 1..n$ with $i \neq j$). As seen above, we consider one distinguished variable, π_i, for each component P_i, to store the current vertex of the TTD of P_i. Hence the set of variables declared in the TTS is:

$$U = U_s \cup \left(\bigcup_{i\in 1..n} U_i\right) \cup \left(\bigcup_{i\in 1..n} \{\pi_i\}\right)$$

Given the parallel composition $\{\Theta\}[\,\|_{i\in 1..n}\, P_i\,]$, we can easily define a TTS with the set of variables U that will give the "semantics of the system". We know that $\langle U_i \cup \pi_i \cup U_s, S_i, s_i^0, \tau_i, I_i\rangle$ is the meaning of P_i for all $i \in 1..n$. Then the meaning of $\{\Theta\}[\,\|_{i\in 1..n}\, P_i\,]$ is the TTS $\langle U, S, s_0, \tau, I\rangle$ (U as defined above) such that:

- the set S lists all the possible interpretation of U,
- the initial state $s_0 \in S$ is the only interpretation such that $s_0(x) = s_i^0(x)$ if $x \in U_i$, $s_0(\pi_i) = v_0^i$ for all $i \in 1..n$, (v_0^i is the initial vertex of the component P_i); and $s_0(x) = \Theta(x)$ for all $x \in U_s$,

– τ is the set of transitions resulting from mapping each edge e in each component P_i to a transition t_e as follows. If e connects vertice v_a^i to v_b^i[1], then $s' \in t_e(s)$ iff
$s(\pi_i) = v_a^i$ and $s'(\pi_i) = v_b^i$;
and $g_e(s)$ is true; and
$s'_{|U_i \cup U_s} = op_e(s_{|U_i \cup U_s})$; and
$s'(x) = s(x)$ for each x in $U \backslash (U_i \cup U_s \cup \{\pi_i\})$.
– I maps every transition t_e to the time interval $I_i(e)$, where P_i is the component containing the edge e.

The notion of a TTS defines a composition operator over TTDs and their compositions. This is basically the same operation as the one in [17] with the simplification that all the components must start in their initial state. That is we only consider a "synchronous start" of TTDs.

4.1.6 Sequential behavior The fact that TTS support the use of variables eases building several classes of systems by simply composing TTDs. We show how to use TTS in order to build a system from the "sequential composition" of components. Sequential composition will be a useful operation when defining the behavior of execution tasks in Sect. 5.

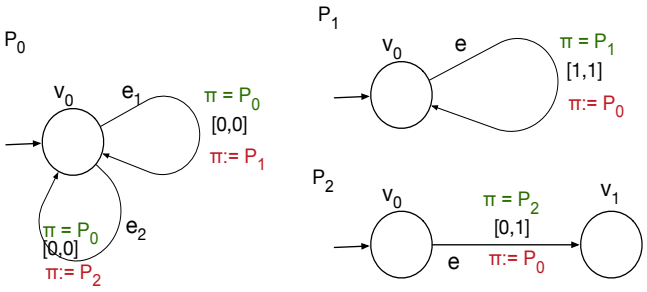


Fig. 4: TTDs of a sequential system

As an example, let us consider the parallel composition of the TTDs in Fig. 4. The set of shared variables U_s contains a variable Π that will denote the "identity" of the only currently executing component; that is $\mathrm{dom}(\Pi) = \{P_0, P_1, P_2\}$ with $\Theta(\Pi) = P_0$.

The sequential composition of the three components is the TTS $\{\Theta\}[\,\|_{i \in 0..2}\, P_i\,]$ where the guard of each edge in the TTD P_i includes the test $\Pi = P_i$. With this constraint, it is only possible to take an edge from the component whose identity is the current value of Π. Therefore, at most one component can execute at a time (no two edges belonging to two different components can be enabled simultaneously).

In this particular example, component P_0 plays the role of a "scheduler" that gives the control randomly to either P_1 or P_2. Giving the control more than once to P_1 leads to a deadlock (no discrete transitions possible in the resulting TTS).

[1] The superscript ^{i} denotes that the vertex is in P_i

4.1.7 Suitability We discuss the rationale for the choice of TTS for formalizing GenoM3, as opposed to *e.g* other formalisms based on clocks, such as timed automata (Sect. 4.2). There are several arguments that favor such a choice among which we emphasize the following.

Variables and compositionality: As seen in Sect. 2, GenoM3 relies on a compositional approach where robotic applications contain several components communicating together. Moreover, components themselves are built from entities that interact in order to ensure a correct behavior with regard to the requirements. For instance, the control task interacts with the execution tasks to instruct them on which activities to run or interrupt (Sect. 3.3). The power of TTS through the composition of TTDs is very useful in such contexts. Indeed, shared variables and the parallel operator ease the modeling of the complex asynchronous communication within and between GenoM3 components. Also, the sequential behavior within execution tasks can be conveniently modeled using shared variables and parallel operators as seen above (more in Sect. 5).

Variables, guards and time intervals: The possibility to have guards over variables and to use time intervals makes TTDs suitable for modeling the entities of a GenoM3 component. For instance, one may, within a TTD model of an activity, condition through the guards the execution of a codel by the availability of resources, and use WCET as upper bounds (more in Sect. 5).

Urgencies: many of the behaviors in GenoM3 are subject to *global urgency constraints* rather than local ones. For instance, executing a codel happens *as soon as* it has secured the needed resources within the IDS, shared between all tasks. These aspects are modeled easily in TTS as opposed to clock-based transition systems such as those resulting from *e.g.* classical timed automata where urgencies can be expressed only locally using *invariants* (Sect. 4.2). The confrontation between the two models in terms of expressing urgencies is explained in details in [11].

4.2 Timed Automata

4.2.1 Introduction Timed Automata (TA) is a theory for modeling and verification of timed systems. In the original version of the theory [3], TA extend *finite-state Büchi automata* with real-valued clocks. The behavior of such automata is therefore restricted by defining constraints on the clock variables and a set of accepting states. A simpler version allowing local invariant conditions and known as *Timed Safety Automata TSA* is introduced in [18]. In this document, we focus on TSA and refer to them as simply TA.

4.2.2 Formal definition A TA is a tuple

$$TA = \langle L, l_0, X, E, I \rangle$$

where:

- L is a finite set of locations,
- $l_0 \in L$ is the initial location,

– X is a finite set of continuous variables called clocks,
– E is a finite set of edges of the form (l, g, e, φ, l'), where l and l' are locations, g is a predicate on $\mathbb{R}^X$ and φ is a binary relation on $\mathbb{R}^X$,
– I assigns an invariant predicate $I(l)$ to any location l.

4.2.3 Semantics The semantics of TA is defined over a Kripke structure, whose states are pairs $s = (l, v) \in L \times \mathbb{R}^X$, with $v \models I(l)$, and transitions defined as:

– delay transitions: $(l, v) \xrightarrow{d} (l, v')$ with $d \in \mathbb{R}_{\geq 0}$ and $v' = v + d$, and
– discrete transitions: $(l, v) \rightarrow l', v')$ if there is an edge (l, g, Y, l') such that $v \models g$ and $v' = v[Y]$, where $Y \subseteq X$, and $v[Y]$ is the valuation assigning 0 when $x \in Y$ and $v(x)$ otherwise.

4.2.4 Example Fig. 5 shows a simple TA with three locations l_0 (initial, denoted with an inside ring), l_1 and l_2 and a clock c. With locations l_1 and l_2 are associated the invariants (in purple) $c \leq 2$ and $c \leq 1$, respectively. This means that whenever l_1 (respect. l_2) is reached, it must be left at most when the valuation of c is equal to 2 (respect. 1). The *reset* actions (in blue) assign the valuation 0 to c when the edges they are associated to are taken (edges from l_0 to l_1 and from l_1 to l_2). The guards (in green) must be satisfied when an edge is taken.

The absence of an invariant on location l_0 makes taking its outgoing edge possible no matter what the valuation of c is. Let τ be the date at which the outgoing edge of l_0 is taken. This means that l_1 is reached at τ and must be left within the interval $]\tau, \tau + 2]$. This interval is left-open because each outgoing edge of l_1 is guarded with the strict inequality $c > 0$ (l_1 cannot be left at τ). Let τ' be the date at which the edge from l_1 to l_2 is taken. Location l_2 will be left within $]\tau', \tau' + 1]$ and the initial location is reached.

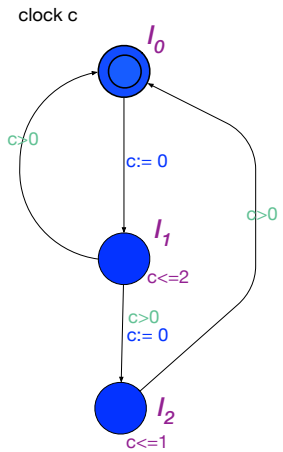


Fig. 5: Timed automaton example

4.2.5 Extending TA

Urgencies TA urgencies may be expressed only locally through invariants. To deal with urgencies expressed globally, *e.g.* involving different TA components, TA are extended with urgencies in [8]. We refer to such formalism as *Urgency Timed Automata UTA*. When an edge in a UTA is *eager*, that we note $\wr$, it *must* be taken (or disabled by taking another edge) as soon as enabled. That is, when an *eager* edge is enabled, time is not allowed to progress and this very edge must be taken or disabled immediately.

Data variables To ease the modeling of real-world systems, often communicating through shared variables, TA may be extended with data variables. In such a case, guards and assignments, originally allowed only on clocks (equality/inequality for guards and reset for assignments), become possible on variables as well. We refer to this extension as DTA. UTA extended with data are referred to as DUTA.

4.2.6 Composition of DUTA The parallel composition of n DUTA is the system $\{Init\}[\,\|_{i\in 1..n}\ A_i\,]$, where each A_i is a DUTA and $Init$ defines the initial valuations of shared variables.

The semantics of a DUTA composition is thus given over a Kripke structure with states of the form of pairs $s = (l, v)$. The difference with the states given in the semantics of a single (non-extended) TA in Sect. 4.2.3 is that now (i) l stores the current location for each DUTA A_i and the valuation of each non-clock variable in the system, (ii) v stores the valuation of all clocks in the composition (in each A_i) and $v \models I_i$ for all A_i.

The transitions are then defined as in Sect. 4.2.3: discrete and delay. Here, the discrete transitions may contain a set of $\wr$ transitions, such that an $\wr$ transition corresponds to an $\wr$ edge. When enabled, an $\wr$ transition deactivates all delay transitions until it is taken (or disabled by taking another concurrent discrete transition). A large example over DUTA compositions in terms of GenoM3 applications is given in Sect. 6.2.

5 Formalizing GenoM3

5.1 Importance and feasibility

As explained in Sect. 1, the absence of formal semantics in low-level robotic frameworks is quite problematic. Indeed, it is mostly cumbersome and error-prone to try to model robotic specifications, written within informal frameworks such as ROS, in formal languages and frameworks. Furthermore, computer science is a mathematically founded discipline where, for instance, formal semantics is at the heart of programming languages. Formal semantics gives a clear, unambiguous definition to the language/-software contrary to informal descriptions that might be interpreted differently by distinct readers. In the case of robotics, such semantics would make it possible to soundly translate robotic specifications into other formalisms. Indeed, since the translated specifications obey some formal semantics, it is possible to construct a proof of soundness between the semantics and the translation (Sect. 6.2).

GenoM3 is amenable to formalization due, mainly, to its model-based nature. Indeed, the definition of the entities a GenoM3 component may have is clear and the set of behavioral rules is finite. For instance, we know that each GenoM3 component has one control task, and we know how it evolves. We also know that a GenoM3 component may have a finite number of activities, and that each activity has a finite number of codels,

and the evolution rules of an activity within its execution task are well defined. Overall, there is a finite set of rules on what the programmer may define (syntactic definitions) and how the component evolves (operational semantics) in GenoM3. This makes the formalization of GenoM3 possible by carefully mapping each entity and rule into TTS.

For readability, we present formal definitions and operational semantics of a *lightweight* version of GenoM3. This version preserves the most important mechanisms including concurrency, mutual exclusion, activation and interruption of activities. Validate codels, control services and aperiodic execution tasks are excluded. These choices permit showing in-depth details on semanticizing ambiguous, yet crucial, software aspects of GenoM3 (such as interruptions). At the same time, the presentation is not overloaded with simpler and clearer mechanisms such as the execution of control services. For simplicity, we abuse notation to make the term *codel* refer, from now on, interchangeably to the codel or the state it is associated to, and bears always the name of the state rather than the function it calls upon execution.

5.2 Syntax and syntactical restrictions

Let $Comp$ be a GenoM3 component. We define hierarchically the constituents of $Comp$:

5.2.1 Activities An activity A is a tuple

$$\langle ID_A, C_A, W_A, T_A, T_A^P \rangle$$

where:

- ID_A is the unique activity name,
- C_A is a set of codels with at least two codels (the entry codel $start_A$ and the final codel $ether_A$):
$$\{start_A, ether_A\} \subseteq C_A.$$
An activity may also have a "stop codel", $stop_A$, that defines the code to be executed when the activity is interrupted,
- $W_A : C_A \backslash \{ether_A\} \mapsto \mathbb{Q}_{>0}$ is a function that associates to every codel its WCET (Sect. 2). We do not define a WCET for the codel $ether$, reserved for termination only (there is no code attached to it),
- T_A is a set of transitions of the form $c \to c'$ where c and c' are codels in C_A. We denote such a transition by simply $c \to$ (or $\to c'$) when the identity of codel c (or c') is unimportant,
- $T_A^P \subseteq T_A$ is the set of *pause* transitions.

5.2.2 Execution task An execution task ET is a tuple

$$\langle Per, \mathcal{A}, Inc, V \rangle$$

where:

- $\mathcal{A}$ is the non-empty set of activities ET is in charge of. We use the notation $ID_{\mathcal{A}}$ to refer to the set $\bigcup_{A \in \mathcal{A}} ID_A$ of all IDs of activities in $\mathcal{A}$,
- $Per \in \mathbb{Q}_{>0}$ is the period,

– Inc is the *incompatibility function* that maps the ID of each activity in $\mathcal{A}$, say ID_A, to the set of activities in $\mathcal{A}$ that are incompatible with A, that is the activities that must be interrupted before A is launched. Therefore:
$Inc : ID_{\mathcal{A}} \mapsto P(ID_{\mathcal{A}})$, where $\mathcal{P}(S)$ denotes the powerset (the set of all subsets) of S,
– V is a set of variables.

5.2.3 Control task A control task CT is a specific task dedicated to the interaction between a component and its surrounding. It is also responsible for "marking" an activity as ready for execution or for interruption, and reports on the termination of activities. In GenoM3, the user does not specify the control task whose behavior is defined implicitly. Therefore, a control task is only defined at this level by a set of local variables that we call V.

5.2.4 Component A component $Comp$ is a tuple $\langle CT, E, V, \mu \rangle$ where:

– CT is a control task,
– E is a set of execution tasks,
– V is a set of variables (shared between CT and each ET in E).
– $\mu : C \mapsto \mathcal{P}(C)$ is the mutual exclusion function, where C is the union of all the codels in all activities of all execution tasks in E. Informally, the set $\mu(c)$ lists the set of codels that cannot simultaneously execute with c. In the remainder of this document, codels c such that $\mu(c) = \varnothing$ are referred to as *thread safe*. Otherwise we say that c is *non thread safe*.

5.2.5 Application and well-formed specifications An application, denoted App in the rest of the text, is simply a set of components.

We will only consider *well-formed* applications, that are defined by syntactic restrictions on the activities and execution tasks that they include.

First, we require that each codel in an activity A, excluding $ether_A$, must have at least one successor in the relation defined by the set of transitions T_A. More formally, for any activity A and codel c in $C_A \setminus \{ether_A\}$, there must be a transition of the form $c \to c'$ in T_A, with $c' \in C_A$.

Second, we require that a transition in T_A must not involve a *stop* codel as a target. Indeed stop codels are reserved for interruptions. Similarly, it cannot involve an *ether* codel as its source, since *ether* is reserved for termination. Also, an *ether* codel cannot be the target of a *pause* transition because the latter is for suspension until the next period, while the former is for termination.

All the previous requirements can be expressed more succinctly with the following constraints:

$$\forall c \in C_A \setminus \{ether_A\}\ \exists c' \in C_A : c \to c' \in T_A$$
$$\forall c, c' \in C_A : (c \to c' \in T_A) \Rightarrow (c \neq ether_A \wedge c' \neq stop_A)$$
$$\forall c, c' \in C_A : (c \to c' \in T_A^P) \Rightarrow (c' \neq ether_A)$$

Finally, $ether$ codels are always thread safe (there is no code attached to them, Sect. 5.2.1). Also, there must be no mutual exclusion between codels of activities that belong to the same execution task. Indeed, any two activities A and B in the same

execution task are executed sequentially "by construction" (no activities in the same task can run concurrently). Therefore we require that $\mu(c) \cap C_B = \mu(c') \cap C_A = \varnothing$ for all c in C_A and c' in C_B.

5.3 Semantics of lightweight GenoM3

The operational semantics of GenoM3 entities is given in terms of TTDs that are composed together to build components and applications. Then we can derive a notion of reduction on GenoM3 by lifting the corresponding relation at the TTS level (Sect. 4.1.3). As a consequence, we can define the behavior of GenoM3 components independently from the implementation (in accordance with the informal description given in Sect. 3.3). In the next section, we refine the operational semantics by defining a more precise notion of actions.

Here, we need to distinguish between what the programmer specifies (which is reflected at the syntactical level, for instance in transitions between codels declared in activities, Sect. 5.2), and what is implicitly specified, that is, enforced at execution to produce the expected behavior, like for instance interruption transitions (to codel `stop` if it exists). Indeed, the programmer does not specify transitions to the `stop` codel, if it exists (Sect. 5.2.5), as such transitions are defined by default and automatically executed when applicable (Sect. 3.3). We define the semantics of a GenoM3 component gradually through three levels:

- Mono-task component: the component contains only one execution task (no control task),
- Multi-task component: the component contains a finite number of execution tasks (no control task),
- All-task component: the component contains a finite number of execution tasks and a control task.

This layering will help us present the semantics progressively, in an understandable way, but also select the right level according to the objective (presentation, translation, proof) such as both readability and convenience are preserved (more in next section).

5.3.1 Level 1: mono-task component This is the lowest level in complexity (and highest abstraction). In this context, the component contains only one execution task, which means that all the codels are thread safe (see the property of $\mu()$ in Sect. 5.2.5).

For the sake of simplicity, we stop referring to the names of edges in TTDs (Sect. 4.1.4). That is, an edge e that connects vertex v to v', denoted also $v \xrightarrow{e} v'$ in Sect. 4.1.4 will be referred to, from now on, as simply $v \rightarrow v'$, $v \rightarrow$ (when the identity of v' is uninmportant), or $\rightarrow v'$ (when the identity of v is unimportant). This will alleviate the notations but still permits to define edges uniquely through their source and target vertices and the set they belong to as we will see hereafter. It will also ease loading edges with *actions* in Sect. 6.2.

Definition 1 ***Activities semantics.***
The operational semantics of an activity $\langle ID_A, C_A, W_A, T_A, T_A^P \rangle$ (Sect. 5.2.1) is given by a TTD (Sect.4.1.4) such that:

- *Vertices V: each $c \in C_A$ is mapped to one vertex with the same name $c \in V$. The initial vertex v_0 is $ether_A$.*

- *Edges E are partitioned into a set of* nominal edges*, E^N, and additional edges, E^A. That is $E = E^N \cup E^A$ where:*
  - *Nominal edges: each transition $c \to c'$ in T_A is mapped to an edge $c \to c'$ in E^N. We distinguish three disjoint sets of nominal edges:*
    $E^N = E^P \cup E^T \cup E^X$. E^P is the (possibly empty) set of pause edges that maps the set of pause transitions T^P; E^T is the (possibly empty) set of termination edges of the form $\to ether$ and E^X the (possibly empty) set of the remaining (execution) edges.
  - *Additional edges: We distinguish two disjoint sets of additional edges:*
    $E^A = E^S \cup E^I$. E^S contains the additional edge for starting $ether \to start$. E^I is the set of additional edges for interruption: (i) from vertex $c = ether$ and (ii) from each vertex c such that there is an edge $\to c$ in E^P to vertex $stop_A$ if $stop_A \in C_A$ (to vertex $ether_A$ otherwise).
- *Time intervals I: $I =]0, W(c)]$ for each edge in E^N and $I = [0, 0]$ for each edge in E^A.*

Consequently, the set of nominal edges maps the transitions that the programmer specifies, while the set of additional edges reflects internal actions enforced by GenoM3 to handle starting and interruption of activities. The additional edges for interruption E^I ensure that an activity that is interrupted before starting or after a pause will execute the interruption routine: transit to `stop` (if it exists) or terminate by transiting to `ether` (otherwise).

Edges uniqueness For activities, due to the restrictions defined in Sect. 5.2.5, the sets E^N and E^A are necessarily disjoint, and thus all subsets of E^N and E^A are mutually disjoint. Moreover, it is not possible to have two different edges with exactly the same source and target codel, so specifying the source and target of an edge defines it uniquely. The only exception is for codels c that are both the target of a pause transition ($\exists \to c \in T_A^P$) and the source of a transition to *ether* ($c \to ether_A \in T_A$) in an activity that does not have a *stop* codel. In this case, we end up with two edges connecting c to $ether_A$: one nominal for termination (in the set E^T) and one additional for interruption (in the set E^I). Here, it is sufficient to mention also to which set the edge belongs to define it uniquely. For TTDs of other entities excluding the control task (such as the task *manager*, see below), edges are uniquely defined through their source and target vertices. This remains true at *level 2* and *level 3* (next sections).

Example This example shows the definition of an activity A and its operational semantics.
Syntactic definition (from Sect. 5.2.1)

- $C_A = \{start_A, main_A, ether_A\}$,
- $W_A(start_A) = 1$, $W_A(main_A) = 2$,
- $T_A = \{start_A \to main_A, main_A \to ether_A\}$,
- $T_A^P = \varnothing$.

Semantics We apply **Definition 1** to A to get the TTD of A in Fig. 6. Note the edge from $ether_A$ to $ether_A$ that represents interruption (absence of codel $stop_A$ here).

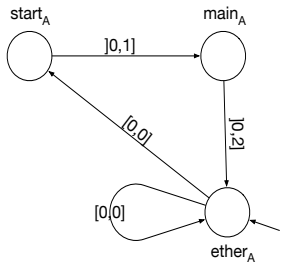


Fig. 6: Activity TTD example (mono-task context)

Definition 2 ***Execution task semantics.***
The semantics of an execution task $ET = \langle Per, \mathcal{A}, Inc, V \rangle$ *is a TTS (parallel composition, Sect.4.1.5)*

$$ET = \{\Theta\}[Tim||Ex]$$

where Θ *gives the initial values of the shared variables (given below) and* Tim *is the timer.*

Ex *is a TTS (sequential composition, Sect.4.1.6)*

$$\{\Theta\}[M||(\underset{A\in\mathcal{A}}{||}A)]$$

where M *is the task manager and* $\underset{A\in\mathcal{A}}{||}A$ *is the sequential composition (Sect.4.1.6) of all activities* A *in* $\mathcal{A}$ *(Sect. 5.2.2).*

The set of variables V *contains:* N*, the set of names of activities to be executed nominally,* R*, the set of names of activities to be interrupted (both* N *and* R *are defined over* $ID_{\mathcal{A}}$*, Sect. 5.2.2),* sig*, the period signal (boolean), and* Π*, the control passing variable (of type* $ID_{\mathcal{A}} \cup M$*, the same idea as in Sect.4.1.6). The initial values are* $\Theta(N) = \Theta(R) = \varnothing$*,* $\Theta(sig) = False$*, and* $\Theta(\Pi) = M$*.*

Π is initialized to M to ensure that the manager has the control when the system starts (the global control is held by the *manager* M at the initial state of the underlying TTS). Both Tim and M are TTDs whose behavior is defined in the sequel.

Definition 3 ***Timer semantics.***
The timer has one vertex and one edge. The latter is associated with the interval $[Per, Per]$ *and the operation* $sig := true$ *(Fig. 7).*

Changing the value of *sig* to *true* corresponds to transmitting a signal asynchronously to the *manager* (see below). The time interval $[Per, Per]$ ensures that this signal is transmitted at exactly each period (each Per time units).

Definition 4 ***Manager semantics.***
The manager is a TTD with two vertices: wait and manage. The edges, guards, operations and time intervals are shown in Fig. 8.

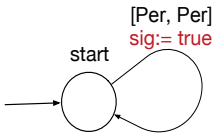


Fig. 7: Timer TTD

The location *wait* denotes waiting for the next period signal and *manage* is to execute activities, if any. The union $N \cup R$ defines the set of activities to execute through their IDs. The operation $\Pi := rand(N \cup R)$ gives the global control randomly to one of the activities whose ID is in $N \cup R$ (by assigning randomly an element from $N \cup R$ to Π). The manager transits back to *wait* as soon as the set defined by this union is empty.

Since $\Theta(N) = \Theta(R) = \varnothing$, no activity would ever be executed by the manager. This is normal because fulfilling activities requests is the role of the control task that we do not have at this level. The manager performs the operation $rrand(N, R)$ to solve this problem. It initializes randomly N and R, over the set of IDs of the activities ET is in charge of; while respecting the disjointness condition $N \cap R = \varnothing$ and the uniqueness condition $(ID_A \in S \wedge ID_B \in S) \Rightarrow (A \neq B)$ with S is either N or R. Note how

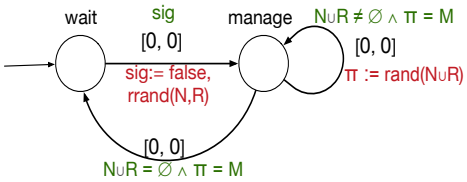


Fig. 8: Manager TTD

the guard on the edge from *wait* to *manage* does not contain the clause $\Pi = M$. This is indeed not necessary as we may easily prove that if the *manager* is at vertex *wait*, then $\Pi = M$ (this kind of invariants will be useful when we prove the soundness of our translation in the next section):

- First visit: $\Theta(\Pi) = M$,
- Subsequent visits: each subsequent visit results from taking the edge from *manage* to *wait*, which is itself guarded by $\Pi = M$ and does not modify Π,
- Time progress: all operations that change Π from M to something else are on the edges whose source vertex is *manage*, which means that the value of Π when reaching *wait*, proven above to be M, will remain so as long as the current vertex of *manager* is *wait*. Activities have also access to Π but never change it to something else than M (see below).

Now we see how the TTD of an activity A given in **Definition 1** is enriched with guards and operations when involved in the execution task.

Definition 5 ***Activities semantics (enriched).***
Each incoming edge to $ether$ (each element of E^T if a codel stop exists, of $E^T \cup E^I$ otherwise) and each pause edge (each element of E^P) is augmented with the operation $\Pi := M$ and the operation $UP(ID, N, R)$ that removes ID from N or R, whichever set it belongs to. Additional edges for interruption (E^I) are guarded with $\Pi = ID \wedge ID \in R$. The starting edge (the only element of E^S) and each edge $c \rightarrow$ in E^N such that there exists an edge $\rightarrow c$ in E^P are guarded with $\Pi = ID \wedge ID \in N$.

Let us illustrate with an example how this augmentation with guards and operations coincides with the behavior given in Sect. 3.3. We consider the same activity A (Fig. 6) and a second activity B defined as follows:

- $C_B = \{start_B, main_B, stop_B, ether_B\}$,
- $W_B(start_B) = 1$, $W_B(main_B) = 2$, $W_B(stop_B) = 1$,
- $T_B = \{start_B \rightarrow main_B, main_B \rightarrow main_B, stop_B \rightarrow ether_B\}$,
- $T_B^P = \{main_B \rightarrow main_B\}$.

We apply **Definition 5** to get the TTDs of A and B in Fig. 9 when evolving within the execution task whose manager and timer are represented in Fig. 8 and Fig. 7 (**Definition 4** and **Definition 3**), respectively. The non-determinism on whether to execute nominally or interrupt at the beginning of the execution (from $ether$ or wherever the activity was paused) is resolved by finding to which set the activity ID belongs (*e.g.* edges from $main_B$ to $stop_B$ and from $main_B$ to $main_B$). At the end of the execution, either by taking a $pause$ edge (*e.g.* edge from $main_B$ to $main_B$) or reaching $ether$ (*e.g.* edge from $main_A$ to $ether_A$), the control is given back to the $manager$ through the operation $\Pi := M$. Together with such operation, the activity updates the set N or R by removing its ID from the set it belongs to through the operation $UP(ID, N, R)$. This is to denote that there is no further execution required for this activity in the current cycle. Note that checking whether the activity has the control is necessary only on starting and interrupting edges and when resuming after a pause (**Definition 5**) as we may easily prove that when activating any of the remaining edges, Π is always equal to the activity ID.

Component At this level, the component is simply the execution task ET. It is thus derived from **Definition 2**.

5.3.2 Level 2: multi-task component At this level, the component may contain several execution tasks, which means that some codels may be non thread safe. Only the operational semantics of activities change.

Definition 6 ***Activities semantics (level 2).***
The operational semantics of an activity $\langle ID_A, C_A, W_A, T_A, T_A^P \rangle$ (Sect. 5.2.1) is given by a TTD such that:

- *Vertices V: each $c \in C_A$ s.t. $\mu(c) \neq \varnothing$ (non thread safe) is mapped to two vertices c and c_{exec}. **Definition 1** applies otherwise.*

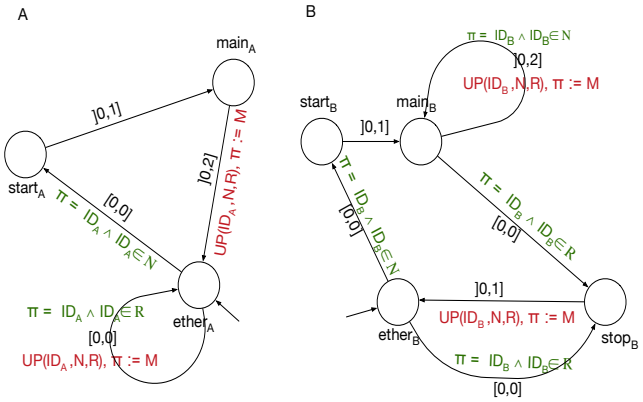


Fig. 9: Activities A and B in task ET (level 1)

- *Edges E: partitioned into nominal edges E^N and additional edges E^A:*
  - *Nominal edges E^N: partitioned into E^P (pause edges), E^T (termination edges) and E^X (execution edges). Each transition in $T_A \backslash T_A^P$ from a non-thread-safe codel c to c' is mapped to an edge $c_{exec} \rightarrow c'$ in E^X (in E^T if $c' = ether$). Each transition in T_A^P from a non-thread-safe codel c to c' is mapped to an edge $c_{exec} \rightarrow c'$ in E^P. For the remaining transitions in T_A, **Definition 1** applies to get their mapping in E^N,*
  - *Additional edges E^A: partitioned into E^I (interruption edges), E^S (starting edges) and E^M (mutual exclusion edges). E^M is the set of edges $c \rightarrow c_{exec}$ for all non-thread-safe codels c. **Definition 1** applies to get E^S and E^I.*
- *Time intervals: **Definition 1** applies on all edges.*

The manager and the timer remain unchanged (**Definition 3** and **Definition 4**). Now we see how the activities at this level are enriched when involved in ET.

Definition 7 ***Activities semantics (enriched, level 2).***
Definition 5 *applies. Then, each additional edge $c \rightarrow c_{exec}$ in E^M is guarded with $Fr(c) \wedge \Pi = ID \wedge ID \in N$ if there exists an edge $\rightarrow c$ in E^P ($Fr(c)$ otherwise), such that $Fr(c)$ is true if and only if c'_{exec} is not the current vertex of its activity (in the global state of the underlying TTS) for all c' in $\mu(c)$.*

The guard $Fr(c)$ is to ensure no two codels sharing some resources run simultaneously. It is implementable through *e.g.* shared variables (see example in [13], section 6.1).

Example Let us consider the same activities A and B from the previous level semantics (Sect. 5.3.1). The behavior is the same, but some codels become non thread safe due to the existence of other execution tasks:

- Activity A: The codel $main_A$ becomes non thread safe ($\mu(main_A) \neq \varnothing$).
- Activity B: The codel $main_B$ becomes non thread safe ($\mu(main_B) \neq \varnothing$).

Applying **Definition 6** then **Definition 7** to A and B give the TTDs in Fig. 10.

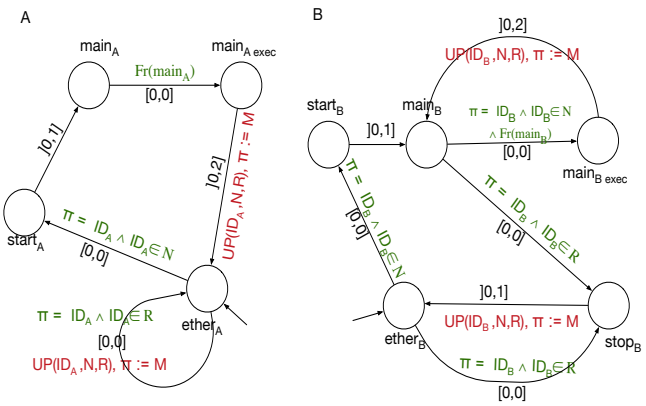


Fig. 10: Activities A and B in task ET (level 2)

Component At this level, the component is the TTS

$$Comp = [E]$$

where $E = \|_{i \in 1..n}\ ET_i$ is the parallel composition of all execution tasks in the component $Comp$.

5.3.3 Level 3: all-task component

Definition 8 ***Control task semantics.***
The semantics of a control task (Sect. 5.2.3) is given over the TTD in Fig. 11 where: $rec(ID)$ *evaluates to true when an activity ID is received,* $Insert(ID, Wa)$ *inserts the received ID in the local variable (a set)* Wa*, and report is the operation of reporting to external entities.*

Requesting an activity, denoted by the guard $rec(ID)$ (ID received), triggers an urgent edge from the initial vertex $idle$ to the vertex $busy$. The received activity name is inserted in the set Wa, which is an initially empty local variable denoting the names of the activities waiting for activation (it is the only element of V given in Sect. 5.2.3). Another possible edge with the same source and target vertices is triggered when an activity finishes its execution (the guard will be formalized later). The edge from $busy$ to end includes the operations of *e.g.* interruption and activation, which will be given later. The edge from end to $idle$ corresponds to sending replies through the operation *report* to external entities.

Definition 9 ***Component semantics.***
a GenoM3 *component is a TTS*

$$Comp = \{\Theta\}[CT \parallel E]$$

where CT *is a control task, and* $E = \|_{i \in 1..n}\ ET_i$ *is the TTS resulting from the parallel composition of all the* n *execution tasks* ET *in component* $Comp$ *(*CT *and* E *are the*

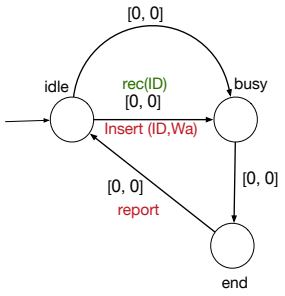


Fig. 11: Control task TTD

operational counterparts of CT and E in Sect. 5.2.4), and Θ gives the initial values of the shared variables (from V in Sect. 5.2.4). These shared variables are: Act the set of activated activities, In the set of interrupted activities and Fi the set of finished activities. $\Theta(Act) = \Theta(In) = \Theta(Fi) = \varnothing$.

Act and In are modifiable only by CT that determines who is activated and who is interrupted (read-only for E) and Fi is modifiable by everyone (both activities and control task need to update it).

Now, the execution tasks and control task diagrams will be enriched with operations over shared variables to ensure a correct behavior within $Comp$ (with regard to that given informally in Sect. 3.3). Within an execution task, the manager (**Definition 4**) and the activities (**Definition 7**) will be enriched as follows:

Definition 10 ***Manager semantics (level 3).***
*On the edge from $wait$ to $manage$ (**Definition 4**), N and R are copied from Act and In, respectively (instead of randomization). Only the names of the activities that this execution task is in charge of (i.e. activities members of $\mathcal{A}$, Sect. 5.2.3) are copied, excluding those in Fi. That is, for task ET, the restricted copy of Act into N results in the set $N = \{ID_A | A \in \mathcal{A} \wedge A \in Act \wedge A \notin Fi\}$ (and similarly when copying In into R). We denote this operation by $rcopy$ (restricted copy), see Fig. 12.*

The restricted copy eliminates possible infinite execution scenarios (the execution task makes the copy once, the activities activated afterwards will be processed at the next period). Excluding the elements in Fi when copying ensures that already terminated activities will not be re-executed (unless requested again in the future).

Definition 11 ***Activities semantics (level 3)***
*The enriched TTD is obtained from **Definition 7**. Then, on each incoming edge to $ether$ (each element of E^T if a codel $stop$ exists, of $E^T \cup E^I$ otherwise), a new operation that inserts the activity ID in the set Fi is added.*

This new operation will notify the control task to act accordingly on the termination of the activity (see below). Applying **Definition 11** to activities A and B (Fig. 10) gives the TTDs in Fig. 13.

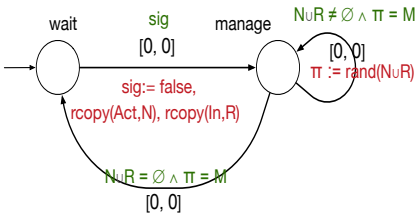


Fig. 12: Manager TTD (level 3)

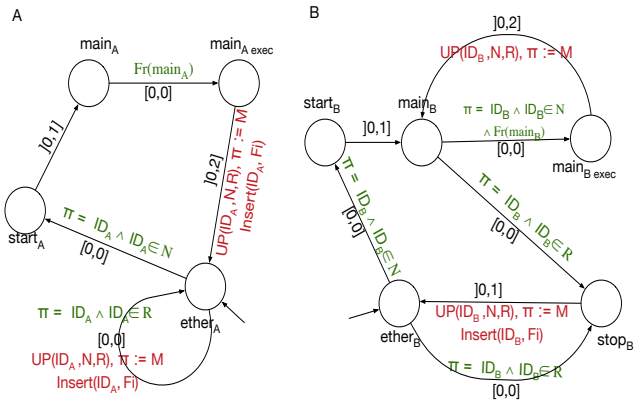


Fig. 13: Activities A and B in task ET (level 3)

Definition 12 ***Control task semantics (enriched).***
*The control task (**Definition 8**) is enriched as follows: the edge $idle \rightarrow busy$ (not guarded with $rec(ID)$) is guarded with a non-emptiness condition on Fi (Fig. 14). The edge $busy \rightarrow end$ is associated with the following operations (in this order):*

- *update Act and In by removing the IDs in Fi:*
  $Act := Act \backslash (Act \cap Fi)$ (and same for In)
  We refer to this operation as $U()$,
- *empty Fi,*
- *Activate and interrupt: move elements of Wa to Act if possible (and from Act to In if necessary):*
  $\forall id \in Wa$:
  ***if** $Inc(id) \cap (Act \cup In) = \varnothing$ then*
  $Wa := Wa \backslash \{id\}$ and $Act := Act \cup \{id\}$ (activation)
  ***else if** $Inc(id) \cap Act \neq \varnothing$ then*
  $In := In \cup (Act \cap Inc(id))$ and $Act := Act \backslash (Act \cap Inc(id))$ (interruption).
  We refer to this operation as $A\_I()$.

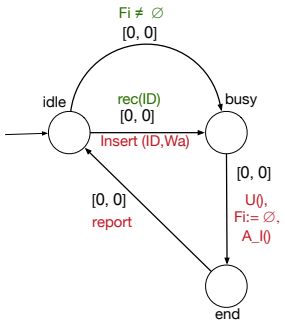


Fig. 14: Control task TTD (enriched)

The guard $Fi \neq \varnothing$, combined with the urgency interval $[0, 0]$ (the edge $idle \rightarrow busy$ not guarded with $rec(ID)$), allows the control task to update the sets Act and In as soon as an activity ends. The operation $U()$ on the edge $busy \rightarrow end$ ensures that this update is correct by removing the ended activities from the sets of activities to be executed ($Act \cup In$). The operation $A\_I()$ activates the waiting activities if possible. That is, for each waiting activity A (in Wa), it checks if there is at least an activity incompatible with it that is still not terminated (in $Act \cup In$). If it is the case, then A needs to wait further (remains in Wa) and the incompatible activities with A that are not interrupted (in Act) need to be moved to In. Otherwise, A is activated (moved from Wa to Act). After these operations, the control task reports to the external entity that requested the finished activities (if any, edge $end \rightarrow idle$).

5.3.4 Application A robotic specification written in GenoM3 contains usually several components. We give thus the definition of a robotic application in terms of operational semantics. We can apply this at any level, which gives us different views of an application at different levels of abstraction. Note that the data flow through *ports* is not specified at this level as its mechanisms depend on the implementation [12].

Definition 13 ***Application semantics.***
An m-component specification is the TTS resulting from the parallel composition of all components

$$app = [||_{i \in 1..m}\ Comp_i]$$

.

6 Translation of GenoM3 Semantics

6.1 Translation to DUTA

DUTA use clocks which evolve monotonically with time and do not depend on edges enabledness. It is thus important to translate while preserving a semantically equivalent behavior under clocks. This equivalence will be proven using bisimulation (Sect. 6.2). From the previous section, we easily notice that the main source of complexity in GenoM3 resides at the execution tasks level. Thus, for readability and convenience, we restrict our translation to the first two levels of operational semantics (Sect. 5.3.1 and

Sect. 5.3.2). At these levels, we use the $rrand(N, R)$ initialization (Sect. 5.3.1) which covers all the possible evolutions of execution tasks as N and R would contain at least all possible IDs if the control task was involved. That is, the set of all the possible configurations of N and R resulting from the application of $rrand(N, R)$ is a superset of that resulting from applying the restricted copy $rcopy(N, R)$ (Sect. 5.3.3).

6.1.1 Mono-task component The objective is now to define the DUTA equivalent to the TTS of ET (Sect. 5.3.1):

$$\{\Theta\}[Tim \parallel M \parallel (\underset{A \in \mathcal{A}}{\parallel} A)]$$

where Tim, M and A are, respectively, the DUTA translations of the timer, the manager and each activity in $\mathcal{A}$. Θ will define the initial values of shared variables in the DUTA of ET that will have the same names as in in the TTS, *i.e.* N, R, Π and sig. We give hereafter the definitions of the elements of the DUTA of ET.

Definition 14 ***Timer Tim (DUTA).***
The DUTA translation of the $timer$ is given by the following rules:

- *clocks: The timer has a clock xt, whose initial valuation is zero,*
- *locations: The timer has one location $start$ that maps the vertex $start$ of its TTD counterpart (**Definition 3**). It is associated with the invariant $xt \leq Per$,*
- *edges: The timer has one edge from $start$ to $start$ that maps its TTD counterpart. With this edge, a guard $xt = Per$ and an operation that resets xt to zero are associated. The $sig := true$ operation is also associated with the same edge.*

The invariant on location $start$ is to enforce its unique outgoing edge to be taken at Per time units at most. The guard on the latter ($xt = Per$) is to ensure taking it at exactly each period, and the reset operation $xt := 0$ to recount the period from zero each time. Consequently, the period signal through sig is sent periodically.

Fig. 15 shows the $timer$ TTD given in **Definition 3** and its DUTA counterpart, resulting from applying **Definition 14**.

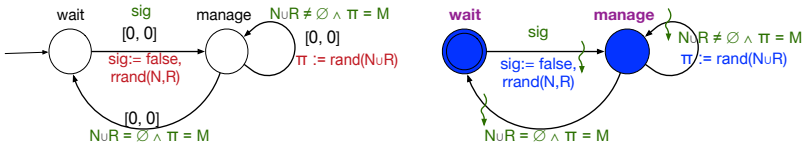
(a) Timer TTD (b) Timer DUTA

Fig. 15: Timer TTD to DUTA (**Definition 14**)

Definition 15 ***Manager M (DUTA).***
The DUTA translation of the $manager$ is given by the following rules:

– *locations: The manager has two locations* $wait$ *and* $manage$ *that map their TTD counterparts (**Definition 4**),*
– *edges: The manager has three edges that map their TTD counterpart. Guards and operations are the same as in the TTD version. Now the urgency on each TTD edge, ensured with* $[0, 0]$ *intervals, is enforced by making each edge in the DUTA counterpart* $eager$*.*

Fig. 16 shows the $manager$ TTD given in **Definition 4** and its DUTA counterpart, resulting from applying **Definition 15**.



(a) Manager TTD (b) Manager DUTA

Fig. 16: Manager TTD to DUTA (**Definition 15**)

We define now translation rules for activities. Due to the special $pause$ statements, one needs to be particularly careful with the translation of activities. For starts, let us consider activity A with the restriction $T_A^P = \varnothing$. We will clarify later with an example why $pause$ behaviors at this level are more delicate to translate to DUTA and propose a solution as a general rule (see **Definition 17** below).

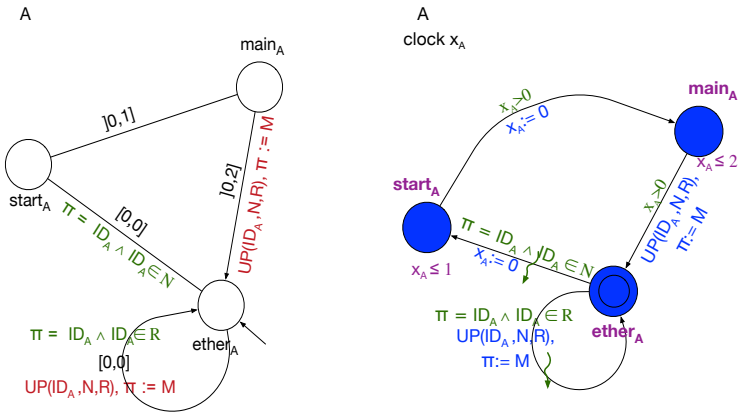
Definition 16 ***Activities A (DUTA, restricted).***
The DUTA translation of an activity $A \models (T_A^P = \varnothing)$ *is given by the following rules:*

– *clocks: An activity* A *has a clock* x_A*, whose initial valuation is zero,*
– *locations: Each vertex in the underlying TTD (**Definition 5**) is mapped to a location with the same name in the DUTA. Each location* $c \neq ether$ *is associated with an invariant* $x_A \leq \uparrow I(c \rightarrow c')$ *with* c' *any vertex in the TTD s.t.* $c \rightarrow c'$ *in* E *(*$\uparrow I$ *of any outgoing edge of* c *is equal to* $W(c)$ *of the underlying codel, **Definition 1**[2]),*
– *edges: (1) Each edge of the underlying TTD is mapped into an edge in the target DUTA with the same source and target. (2) Urgency intervals* $[0, 0]$ *are mapped into* $\wr$ *tags (*$eager$ *edges). (3) Each outgoing edge of a location that is associated with an invariant* $x_A \leq W(c)$ *is guarded with* $x_A > 0$*. (4) Each incoming edge to a location with an invariant* $x_A \leq W(c)$ *is associated with the reset operation over* x_A*. (5) Guards (respect. operations) associated with each edge in the DUTA result from the conjunction (respect. sequencing) of guards (respect. operations) of its TTD counterpart and the guards (respect. resets) of clocks as defined in (3) and (4).*

[2] This is true for all outgoing edges of c here because no pause transition exists in the underlying activity, and thus no interruption is possible from any $c \neq ether$

The invariants ensure that the execution of each codel takes between zero and $W(c)$ units of time. For clock x, the guards $x > 0$ are to eliminate 0 as a possible execution time and the reset operations are to ensure counting $W(c)$ starting at zero. Consequently, each codel c is executed in a non-zero amount of time inferior or equal to its WCET $W(c)$.

As an example, Fig. 17 shows the TTD of activity A (Sect. 5.3.1, Fig. 9 left) and its DUTA counterpart, resulting from applying **Definition 16**.



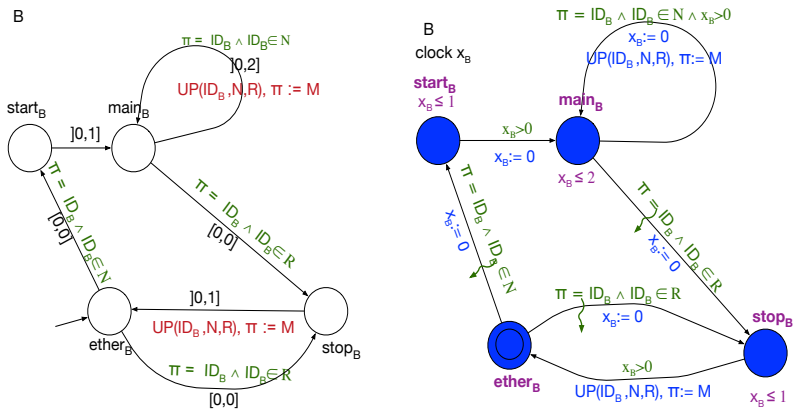
(a) TTD of Activity A task ET (level 1) (b) DUTA of Activity A in task ET (level 1)

Fig. 17: Activities TTD to DUTA (activity A, level 1, **Definition 16**)

Let us now focus on activity B at the same level (Sect. 5.3.1, Fig. 9 right). We note immediately that B violates the restriction in **Definition 16** since $T^P \neq \varnothing$. The activity B is a good practical example to show why **Definition 16** may lead to incorrect translations in some cases due to the nature of clocks in DUTA.

Fig. 18 shows the TTD of activity B (Sect. 5.3.1, Fig. 9 right) and its DUTA counterpart, resulting from applying **Definition 16**. This translation is incorrect. Indeed, if B passes the control back to the $manager$ after a $pause$ transition (taking the edge from $main_B$ to $main_B$ in the DUTA in Fig. 18), the clock x_B will continue evolving monotonically and the DUTA will timelock after 2 time units unless it resumes the control before then (all outgoing edges from location $main_B$ are disabled). This problem is due in part to the fact that clocks evolve independently from edges enabledness in DUTA (in contrast to TTDs where time intervals are relative to the date their edge was last enabled). We propose thus a new generic translation that is valid for all activities at this level without restrictions.

Definition 17 ***Activities*** A ***(DUTA, level 1).***
The DUTA of an activity A *is defined using the following translation rules:*



(a) TTD of Activity B task ET (level 1) (b) Incorrect DUTA of Activity B (level 1)

Fig. 18: Incorrect TTD to DUTA translation (activity B, level 1, **Definition 16**)

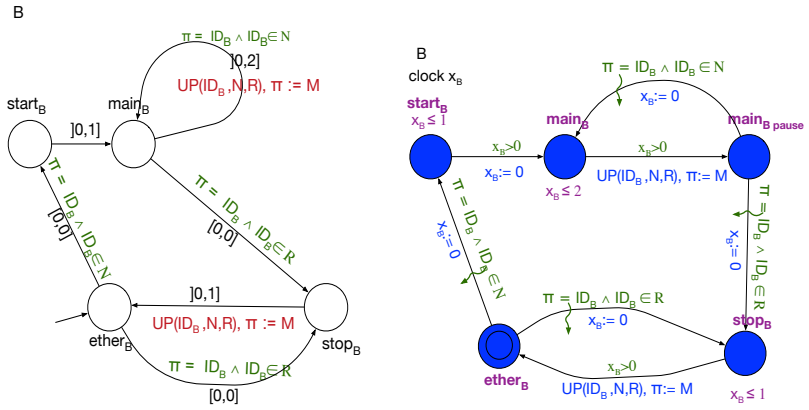
- *clocks: Same as in **Definition 16**,*
- *locations: Each vertex c of a codel c s.t. there exists $\rightarrow c$ in T^P is mapped to, besides the location c (**Definition 16**), another location c_{pause}. The rules on translating vertices in **Definition 16** apply on the remaining vertices to obtain the remaining locations,*
- *edges: Each edge $c \xrightarrow{g,op} c'$ in E^P (**Definition 1**) is mapped to an edge $c \xrightarrow{true,op} c'_{pause}$ in the DUTA, and an eager edge $c'_{pause} \xrightarrow{g,null} c'$ is added ($null$ = no operation). Each interruption edge (in E^I) in the TTD from $c \neq ether$ to $stop$ (respect. to $ether$, **Definition 1**) is mapped to an edge from location c_{pause} to $stop$ (respect. to $ether$)[3]. Rule (1) of **Definition 16** is then applied on the remaining edges of the TTD to obtain the remaining edges of the DUTA. Finally, rules (2) to (5) in **Definition 16** are subsequently applied to all edges.*

These additional rules will allow time on clocks to evolve unboundedly at locations c_{pause}, that is when the activity is paused. Resuming the activity nominally is then equivalent to taking the eager edge $c_{pause} \rightarrow c$ and the clock will be reset at this very edge to count the WCET of c starting from 0.

Now, applying **Definition 17** to activity A will give exactly the same outcome as when applying **Definition 16** (Fig. 17). Let us apply **Definition 17** to activity B for which **Definition 16** is not valid as shown in Fig. 18. The new translation is given in Fig. 19. Here we know that $main_B$ is reached only when B has the control and with a prior clock reset, which eliminates the potential timelock seen in Fig. 18.

6.1.2 Multi-task component The DUTA translation rules remain unchanged for the timer Tim' and manager M'. We extend now translation rules for activities to take into account non-thread-safe codels.

[3] To ensure interruption of a paused activity occurs as soon as the latter is resumed.



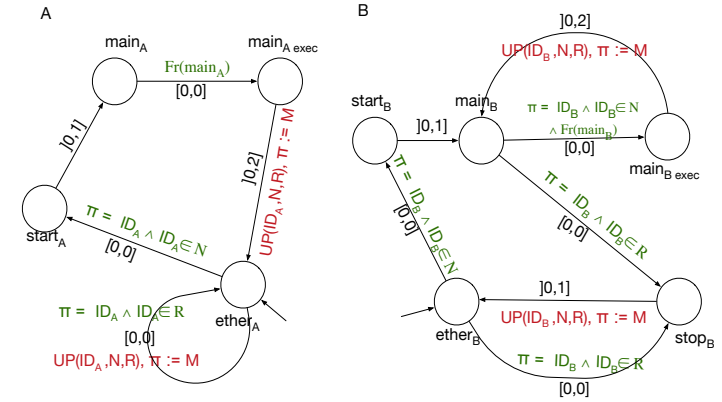
(a) TTD of Activity B task ET (level 1) (b) DUTA of Activity B in task ET (level 1)

Fig. 19: TTD to DUTA translation (activity B, level 1, **Definition 17**)

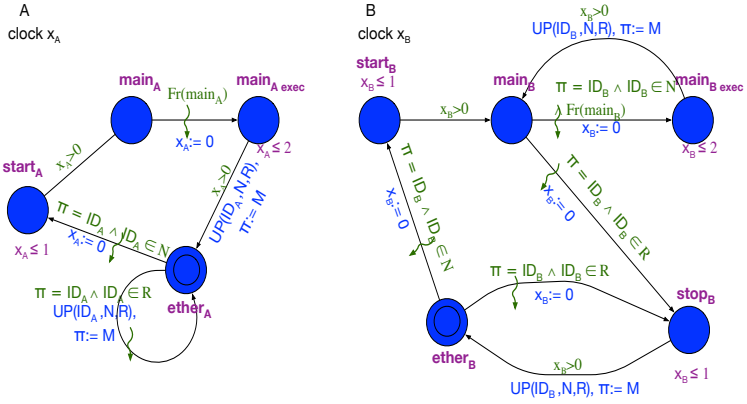
Definition 18 ***Activities*** A' ***(DUTA, level 2).***
The DUTA of an activity A *is defined using the following translation rules:*

- *clocks: Same as in* ***Definition 16****,*
- *locations: Each vertex* c *in the underlying TTD (****Definition 7****) of a thread-safe codel* c *s.t. there exists* $\rightarrow c$ *in* T^P *is mapped to, besides the location* c *, another location* c_{pause}*. Each remaining vertex in the underlying TTD (****Definition 7****) is mapped to a location with the same name in the DUTA. Each location* c *that maps a vertex* c *of a thread-safe codel* $c \neq ether$ *is associated with an invariant* $x_A \leq \uparrow I(c \rightarrow c')$ *with* c' *any vertex in the TTD s.t.* $c \rightarrow c'$ *in* E^N*. The same invariant rule is applied to each location* c_{exec}*,*
- *edges: Each pause edge* $c \xrightarrow{g,op} c'$ *s.t.* c' *is thread safe is mapped to an edge* $c \xrightarrow{true,op} c'_{pause}$ *in the DUTA, and an eager edge* $c'_{pause} \xrightarrow{g,null} c'$ *is added. Each interruption edge in the TTD from* $c \neq ether$ *(whose underlying codel* c *is thread safe) to* $stop$ *(respect. to* $ether$*) is mapped to an edge from location* c_{pause} *to* $stop$ *(respect. to* $ether$*). Rule (1) of* ***Definition 16*** *is then applied on the remaining edges of the TTD to obtain the remaining edges of the DUTA. Finally, rules (2) to (5) in* ***Definition 16*** *are subsequently applied to all edges.*

We note immediately the resemblance between this translation and that given for *level 1*. Indeed, only thread-safe codels targeted by pause transitions induce a non-direct mapping of vertices and edges, and this aspect is already covered at *level 1*. For instance, applying **Definition 18** to activities A and B at level 2 (Sect. 5.3.2, Fig. 10) gives the models in Fig. 20. Notice how, in the absence of thread-safe codels targeted by pause transitions, the translation is rather a one-to-one mapping (besides clock-related constraints).



(a) TTDs of Activities A and B in task ET (level 2)



(b) DUTA of Activities A and B in task ET (level 2)

Fig. 20: Activities TTD to DUTA (A and B, **Definition 18**)

6.2 Translation soundness

We use weak timed bisimulation (**Definition 20** below) to prove that our translation from TTS to DUTA is correct. To make the proof readable and the definitions minimal, we restrict it at *level 1*. This choice is both convenient and representative since it shows the most delicate aspect of the translation, related to thread-safe codels targeted by *pause* transitions. Indeed, we saw previously how, except this aspect, the translation is rather straightforward.

6.2.1 Execution actions To ease following the events within a GenoM3 execution task, we define a set of possible *actions*. Each action represents a category of similar events that obey the same guards and have similar side effects on global variables. This

will also ease reasoning on the soundness of the translation to DUTA. We first define the actions for the original system (in TTS) then the translation (in DUTA).

Nominal execution Nominal edges E^N are the ones explicitly specified in the GenoM3 specification (Sect. 5.2.1 and **Definition 1**). In order to partition these edges according to the actions they pertain to, we need to have a similar precondition for them. The issue here is that nominal edges (members of E^N) do not necessarily obey the property $ID_A \in N$. Indeed, in activity B for instance (Sect. 5.3.1, Fig. 9 right), the edge from $stop_B$ to $ether_B$ is nominal, yet it is taken when $ID_A \in R$. This will make it hard to express nominal actions distinguishably from interruption ones. We propose thus the following.

Definition 19 ***Augmenting interruption edges.***
*Enriching an activity A TTD is given by **Definition 5**, then each interruption edge $c \rightarrow stop$ (in E^I) is augmented with the operation $R := R \backslash \{ID_A\}$ (remove ID_A from R)[4]. The DUTA of A is then obtained from **Definition 17**.*

Lemma 1 ***Correctness of Definition 19.***
*Activities TTDs and DUTA obtained from **Definition 19** induce the same behavior as the ones obtained from **Definition 5** and **Definition 17**. That is, augmenting interruption edges $c \rightarrow stop$ with the operation $R := R \backslash \{ID_A\}$ does not alter the behavior of the execution task.*

Proof. Removing ID_A from R at the beginning of the interruption (when taking the interruption edge) is equivalent to removing ID_A from R at the end of the interruption (with a termination or a pause edge). Indeed, between these two events, A has the control, that is $\Pi = A$, which means that all edges in the *manager* and other activities are disabled (the composition of the activities and the manager is sequential, **Definition 2**). It follows that no edge depending on R is enabled, and thus the behavior remains unchanged.

Additionally, when performing $R := R \backslash \{ID_A\}$ (when taking the interruption edge to $stop$) is followed by performing $UP(ID_A, N, R)$ (when taking a termination edge), removing ID_A from R is redundant, that is the operation $UP(ID_A, N, R)$ is side-effect free (since ID_A has been already removed from R).

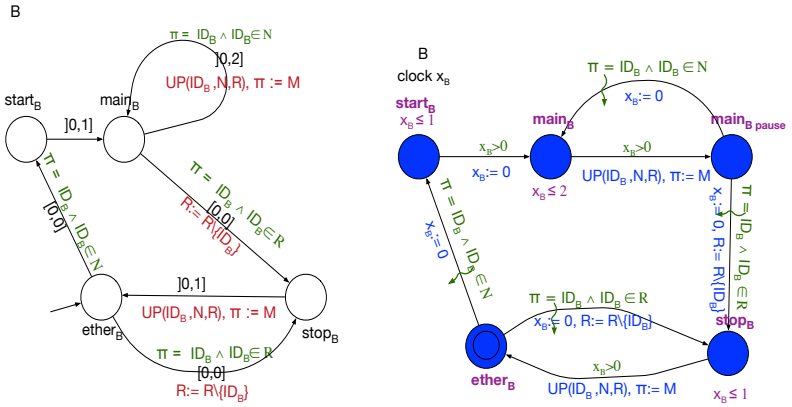
Definition 19 makes it easier to differ between interruption edges and nominal edges. Simply, a nominal edge must satisfy $ID_A \notin R$ while an interruption edge must satisfy $ID_A \in R$. We will use thus this definition for our proof. Fig. 21 shows the TTD and DUTA of activity B (Fig. 19) when applying **Definition 19**.

TTS

First, we partition the edges within an activity as follows:

- Interrupt activity A (ia): This action contains all additional edges for interruption (**Definition 1**), that is all edges in E^I,
- Finish activity A (fa): This action contains all nominal termination and pause edges (**Definition 1**), that is all edges in $E^P \cup E^T$,

[4] if $ether$ is the target codel of the interruption edge, then this is not needed.



(a) TTD of activity B (**Definition 19**) (b) DUTA of activity B (**Definition 19**)

Fig. 21: TTD and DUTA of activity B (**Definition 19**)

– Execute activity A (ea): This action contains all nominal non-pause, non-termination edges plus the additional edge (for starting) $ether \rightarrow start$ (**Definition 1**), that is all edges in $E^X \cup E^S$.

Second, each edge in the manager and the timer corresponds to a distinguished action:

– Start timer (st): corresponds to taking the only possible edge in the *timer* (**Definition 3**),
– Start manager (sm): corresponds taking the edge from vertex *wait* to vertex *manage* (**Definition 4**),
– Launch manager (lm): matches taking the edge from vertex *manage* to itself (**Definition 4**),
– Finish manager (fm): matches taking the edge from vertex *manage* to vertex *wait* (**Definition 4**).

It is intuitive to say that these actions are (i) disjoint and (ii) cover all the edges in the execution task. Indeed, from the partitioning of the actions over edges above and from **Definition 1**, **Definition 3** and **Definition 4**, it follows that the actions cover all the possible edges (no edge remains untied to an action). Additionally, from the definition of the actions above and the mutual disjointness of all the subsets of nominal and interruption edges given in **Definition 1** (Sect. 5.3.1), it follows that the sets of actions are disjoint.

Now, we give for each action some inference rules in terms of TTS semantics: the properties that must be satisfied before taking the action and the side effects of taking it on state variables (and on future dates of taking edges, when uniquely defined). We recall that M and Tim are, respectively, the manager and timer TTDs. By abuse of notation, we refer to an edge by the action it is associated with. For instance, $c \xrightarrow{fa} c'$ is an edge associated with fa (by abuse of notation, an edge fa) from c to c', that is an edge $c \rightarrow c'$ that belongs to $E^P \cup E^T$ (see partitioning of actions above). The edges preserve thus their uniqueness according to their source and target vertex, and

$$st: \frac{0 \in \phi(st) \quad s'(sig) = true \quad \phi'(st) = [Per, Per])}{(s, \phi) \xrightarrow{st} (s', \phi')}$$

Table 2: Action st.

$$sm: \frac{s(sig) = true \quad s(\pi_M) = wait \quad s'(sig) = false \quad s'(\pi_M) = manage \quad s'(N) = N' \quad s'(R) = R'}{(s, \phi) \xrightarrow{sm} (s', \phi')}$$

Table 3: Action sm.

the set of edges they belong to (that we can retrieve from the action on the edge). This simplification helps writing the inference rules without loading the notations further.

Discrete actions (TTS): In the following, s' agrees with s on all state variables unless indicated otherwise. The formula $\exists c \xrightarrow{act} c'$ means there is an edge act from c to c' in the TTD of activity A (even if not enabled). R' and N' are the results of applying $rrand()$ to R and N, respectively.

Action st Taking this action requires satisfying the timing constraints at the timer edge. That is, st is taken at state s in the underlying TTS *iff* the Kripke state (s, ϕ) (see TTS semantics in Sect. 4.1.3) satisfies $0 \in \phi(st)$. Similarly, the state s' satisfies $sig = true$ (table 2).

Action sm To take this action, the manager must be at vertex $wait$ and must have the period signal ($sig = true$). After taking this action, the manager is at location $manage$, sig becomes false and N and R are randomly initialized (table 3).

Action lm To take this action, the manager must have the control ($\Pi = M$) and there must be activities to execute ($(N \cup R) \neq \varnothing$). According to the target Kripke state of this action, we distinguish four cases (table 4): the activity that will take the control is to interrupt from $ether$ (rule $lm.1$), the activity that will take the control is to interrupt after a pause (rule $lm.2$), the activity that will take the control is to execute nominally from $ether$ (rule $lm.3$), or the activity that will take the control is to execute nominally after a pause (rule $lm.4$).

Action fm To take this action, the manager must have the control ($\Pi = M$), must be at vertex $manage$ and there must be no remaining activities to execute ($(N \cup R) = \varnothing$). Taking this action switches the manager vertex to $wait$ (table 5).

Action ia We distinguish four cases (table 6): the source vertex is $ether$ and the target vertex is $stop$ (rule $ia.1$),the source vertex is not $ether$ and the target vertex is $stop$ (rule $ia.2$), the source vertex is $ether$ and the target vertex is $ether$ (rule $ia.3$), the source vertex is not $ether$ and the target vertex is $ether$ (rule $ia.4$).

Action fa We distinguish two cases (table 7): taking a pause edge (in E^P, rule $fa.1$) or taking a termination edge (in E^T, rule $fa.2$).

$$lm.1: \frac{s(\Pi) = M \quad (s(N) \cup s(R)) \neq \varnothing \quad s'(\Pi) = ID_{A \in \mathcal{A}} \quad s'(\pi_A) = ether \quad ID_A \in s'(R)}{(s, \phi) \xrightarrow{lm} (s', \phi')}$$

$$lm.2: \frac{s(\Pi) = M \quad (s(N) \cup s(R)) \neq \varnothing \quad s'(\Pi) = ID_{A \in \mathcal{A}} \quad s'(\pi_A) = c \neq ether \quad ID_A \in s'(R)}{(s, \phi) \xrightarrow{lm} (s', \phi')}$$

$$lm.3: \frac{s(\Pi) = M \quad (s(N) \cup s(R)) \neq \varnothing \quad s'(\Pi) = ID_{A \in \mathcal{A}} \quad s'(\pi_A) = ether \quad ID_A \notin s'(R)}{(s, \phi) \xrightarrow{lm} (s', \phi')}$$

$$lm.4: \frac{s(\Pi) = M \quad (s(N) \cup s(R)) \neq \varnothing \quad s'(\Pi) = ID_{A \in \mathcal{A}} \quad s'(\pi_A) = c \neq ether \quad ID_A \notin s'(R) \quad (\phi'(ea) = I_{ea} \vee \phi'(fa) = I_{fa})}{(s, \phi) \xrightarrow{lm} (s', \phi')}$$

Table 4: Action lm.

$$fm: \frac{s(\Pi) = M \quad (s(N) \cup s(R)) = \varnothing \quad s(\pi_M) = manage \quad s'(\pi_M) = wait}{(s, \phi) \xrightarrow{fm} (s', \phi')}$$

Table 5: Action fm.

Action ea We distinguish two cases (table 8): taking an additional (starting) edge (in E^S, rule $ea.1$) or taking a nominal edge (in E^X rule $ea.2$).

Note that the definitions of discrete actions preconditions are both necessary and sufficient, and also expressed minimally. For instance, the condition to take the action lm (table 4) seems to lack the clause ($\pi_M = manage$). this clause is, however, not necessary because we may easily prove that if $(\Pi = M \wedge (N \cup R) \neq \varnothing)$ then $\pi_M = manage$.

Proof. If $s \models (\Pi = M \wedge \pi_M \neq manage)$ then:
either $s = s_0$, which means that $(N \cup R) = \varnothing)$
or s is reached by taking the edge $manage \xrightarrow{g=(\Pi=M \wedge N \cup R=\varnothing), op=null)} wait$, which means also that $(N \cup R) = \varnothing)$
It follows that the only vertex where $(\Pi = M \wedge (N \cup R) \neq \varnothing)$ may evaluate to true is $manage$

Another example is the lack of the clause $\exists ether \xrightarrow{ea} start$. This is a rule optimization since, by definition (i) any activity has the codels $start$ and $ether$ (Sect. 5.2.1), (ii)

$$\exists ether \xrightarrow{ia} stop$$

$$ia.1: \frac{s(\Pi) = ID_{A \in \mathcal{A}} \quad s(\pi_A) = ether \quad ID_A \in s(R) \quad s'(\pi_A) = stop \quad ID_A \notin s'(R) \quad \phi'(ea) = I_{ea} \vee \phi'(fa) = I_{fa}}{(s, \phi) \xrightarrow{ia} (s', \phi')}$$

$$\exists c \neq ether \xrightarrow{ia} stop$$

$$ia.2: \frac{s(\Pi) = ID_{A \in \mathcal{A}} \quad s(\pi_A) = c \quad ID_A \in s(R) \quad s'(\pi_A) = stop \quad ID_A \notin s'(R) \quad \phi'(ea) = I_{ea} \vee \phi'(fa) = I_{fa}}{(s, \phi) \xrightarrow{ia} (s', \phi')}$$

$$\exists ether \xrightarrow{ia} ether$$

$$ia.3: \frac{s(\Pi) = ID_{A \in \mathcal{A}} \quad s(\pi_A) = ether \quad ID_A \in s(R) \quad s'(\pi_A) = ether \quad s'(\Pi) = M \quad \neg(ID_A \in s'(N) \vee ID_A \in s'(R))}{(s, \phi) \xrightarrow{ia} (s', \phi')}$$

$$\exists c \neq ether \xrightarrow{ia} ether$$

$$ia.4: \frac{s(\Pi) = ID_{A \in \mathcal{A}} \quad s(\pi_A) = c \quad ID_A \in s(R) \quad s'(\pi_A) = ether \quad s'(\Pi) = M \quad \neg(ID_A \in s'(N) \vee ID_A \in s'(R))}{(s, \phi) \xrightarrow{ia} (s', \phi')}$$

Table 6: Action ia.

$$\exists c \xrightarrow{fa} c' \neq ether$$

$$fa.1: \frac{s(\Pi) = ID_{A \in \mathcal{A}} \quad s(\pi_A) = c \quad ID_A \notin s(R) \quad \phi(fa) = I_{fa} - \theta \quad \theta > 0 \quad s'(\pi_A) = c' \quad s'(\Pi) = M \quad \neg(ID_A \in s'(N) \vee ID_A \in s'(R))}{(s, \phi) \xrightarrow{fa} (s', \phi')}$$

$$\exists c \xrightarrow{fa} ether$$

$$fa.2: \frac{s(\Pi) = ID_{A \in \mathcal{A}} \quad s(\pi_A) = c \quad ID_A \notin s(R) \quad \phi(fa) = I_{fa} - \theta \quad \theta > 0 \quad s'(\pi_A) = ether \quad s'(\Pi) = M \quad \neg(ID_A \in s'(N) \vee ID_A \in s'(R))}{(s, \phi) \xrightarrow{fa} (s', \phi')}$$

Table 7: Action fa.

there is always an additional edge $ether \rightarrow start$ (in E^S, **Definition 1**) and (iii) this edge is necessarily an ea edge (see the partitioning of actions above).

Similarly, side effects are expressed minimally and effects on future times to take transitions are mentioned only when certain. For instance, we may easily prove that each execution action ea results in a state where only ea or fa are possible (table 8).

$$ea.1: \frac{\begin{array}{cc} s(\Pi) = ID_{A \in \mathcal{A}} & ID_A \notin s(R) \quad s(\pi_A) = ether \\ s'(\pi_A) = start & \phi'(ea) = I_{ea} \vee \phi'(fa) = I_{fa} \end{array}}{(s, \phi) \xrightarrow{ea} (s', \phi')}$$

$$ea.2: \frac{\begin{array}{c} \exists c \neq ether \xrightarrow{ea} c' \\ s(\Pi) = ID_{A \in \mathcal{A}} \quad ID_A \notin s(R) \quad s(\pi_A) = c \quad \phi(ea) = I_{ea} - \theta \quad \theta > 0 \\ s'(\pi_A) = c' \qquad\qquad \phi'(ea) = I_{ea} \vee \phi'(fa) = I_{fa} \end{array}}{(s, \phi) \xrightarrow{ea} (s', \phi')}$$

Table 8: Action ea.

Proof. From the partitioning, action ea is taken either on a starting edge ($ether \rightarrow start$) or on a nominal edge that is neither a pause nor a termination edge ($c \rightarrow c' \in E^X$). It follows that ea is operation free (no side effects on shared variables (**Definition 1** and **Definition 5**). Now, since $s \models (ID_A \notin s(R))$ (table 8), then $s' \models (ID_A \notin s'(R))$, which means that ia is not possible from s'. Additionally, since each vertex has at least one successor (**Definition 1** and Sect. 5.2.5), then either ea or fa are possible at s'.

Example: In activity B (Fig. 19a), the edge from $main_B$ to $main_B$ is a *pause* edge, it corresponds thus to the ***finish B*** action fb. The same action is associated with taking the termination edge from $stop_B$ to $ether_B$. The edges from $main_B$ to $stop_B$ and $ether_B$ to $stop_B$ are interruption edges, and therefore correspond to the ***interrupt B*** action ib. Finally, the remaining edges are the starting edge and nominal edges that are neither for termination nor for pause, that is ea edges. Fig. 22 shows the TTD of B where edges are tagged with their corresponding actions (guards and operations are omitted for readability).

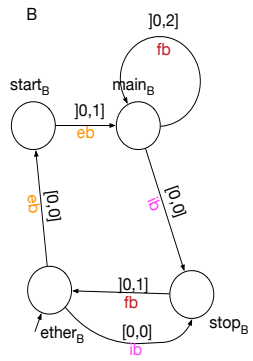


Fig. 22: Actions in activity B

$$d.1: \frac{\phi(st) = I_{st} - a \quad a < Per \quad s(sig) = false \quad s(\pi_M) = wait \quad d \in]0, Per - a]}{(s, \phi) \xrightarrow{d} (s', \phi')} \quad \phi'(st) = \phi(st) - d$$

$$d.2: \frac{\phi(st) = I_{st} - a \quad a < Per \quad s(\Pi) = ID_A \quad s(\pi_A) = c \neq ether \quad ID_A \notin s(R) \quad \phi(ea) = I_{ea} - b \vee \phi(fa) = I_{fa} - b \quad b < W(c) \quad d \in]0, min(Per - a, W(c) - b)]}{(s, \phi) \xrightarrow{d} (s', \phi')} \quad \phi'(st) = \phi(st) - d \quad \phi'(ea) = \phi(ea) - d \vee \phi'(fa) = \phi(fa) - d$$

Table 9: Time actions d.

Time actions (TTS) We define the inference rules of taking a time action in table 9.

Informally, to let time evolve for a strictly positive amount d (non-trivial time step), we must have: (i) A timer period still has not elapsed since the last tick, that is $\phi(st) = I_{st} - a \neq [0, 0]$. Additionally, there must be no urgent edges possible either in the manager M or in an activity A, whichever has the control. If M has the control, that is $\Pi = M$, then the only case where no urgent edge is enabled is when M is at location $wait$[5] given that sig evaluates to false (see the manager model in **Definition 3**). If A has the control, that is $\Pi = ID_A$, it must be at a vertex c different than $ether$ because all possible edges at $ether$ are urgent (see the activities model in **Definition 4**), the urgent edge ia, if exists, must be deactivated at c, that is $ID_A \notin R$ (table 6) and the time elapsed since visiting c must be inferior than $W(c)$ of the underlying codel c (that is $\phi(ea) = I_{ea} - b \neq [0, 0] \vee \phi(fa) = I_{fa} - b \neq [0, 0]$). (ii) The time amount to let elapse must be superior to zero (non trivial) and must not violate any timing constraint, that is it must be at most equal to $\uparrow\phi(st)$ (if the manager is at $wait$) or the supremum of $\uparrow\phi(st)$, $\uparrow\phi(ea)$, and $\uparrow\phi(fa)(otherwise)$.

DUTA The composition of the DUTA of the *manager* M, *timer* Tim and activities (**Definition 14**, **Definition 15** and **Definition 18**) results in a Kripke structure with pairs (l, v) as states (Sect. 4.2). We may thus define the conditions and side effects for each action, defined in the original TTS, in the DUTA system.

Actions (DUTA translation). In the following, l' agrees with l on all state variables unless indicated otherwise. R' and N' are the results of applying $rrand()$ to R and N, respectively. We keep the notation $\pi(P)$, used in TTS, to denote the control location of DUTA P. The inference rules for actions st, sm, lm, fm, ia, fa and ea are given, respectively, in table 10, table 11, table 12, table 13, table 14, table 15 and table 16.

Example: In activity B (Fig. 19b), the edge from $main_B$ to $main_{B\,pause}$ maps a *pause* edge in its TTD counterpart, it corresponds thus to the *finish B* action fb. The interruption action ib is associated with taking any of the edges $main_{B\,pause}$ to $stop_B$ or $ether_B$ to $stop_B$, as both map interruption edges in the TTD counterpart. The edge $ether_B$ to $start_B$ and the edge $start_B$ to $main_B$ map, respectively, the starting edge

[5] Here $\Pi = M$ is redundant since the manager is at location $wait$, hence the absence of the precondition $\Pi = M$ from table 9.

$$st: \frac{v(xt) = Per \quad l'(sig) = true \quad v'(xt) = 0}{(l, v) \xrightarrow{st} (l', v')}$$

Table 10: Action st (DUTA).

$$sm: \frac{l(sig) = true \quad l(\pi_M) = wait \quad l'(sig) = false \quad l'(\pi_M) = manage \quad l'(N) = N' \quad l'(R) = R'}{(l, v) \xrightarrow{sm} (l', v')}$$

Table 11: Action sm (DUTA).

$$lm.1: \frac{l(\Pi) = M \quad (l(N) \cup l(R)) \neq \varnothing \quad l'(\Pi) = ID_{A \in \mathcal{A}} \quad l'(\pi_A) = ether \quad ID_A \in l'(R)}{(l, v) \xrightarrow{lm} (l', v')}$$

$$lm.2: \frac{l(\Pi) = M \quad (l(N) \cup l(R)) \neq \varnothing \quad l'(\Pi) = ID_{A \in \mathcal{A}} \quad l'(\pi_A) = c_{pause} \quad ID_A \in l'(R)}{(l, v) \xrightarrow{lm} (l', v')}$$

$$lm.3: \frac{l(\Pi) = M \quad (l(N) \cup l(R)) \neq \varnothing \quad l'(\Pi) = ID_{A \in \mathcal{A}} \quad l'(\pi_A) = ether \quad ID_A \notin l'(R)}{(l, v) \xrightarrow{lm} (l', v')}$$

$$lm.4: \frac{l(\Pi) = M \quad (l(N) \cup l(R)) \neq \varnothing \quad l'(\Pi) = ID_{A \in \mathcal{A}} \quad l'(\pi_A) = c_{pause} \quad ID_A \notin l'(R)}{(l, v) \xrightarrow{lm} (l', v')}$$

Table 12: Actions lm (DUTA).

$$fm: \frac{l(\Pi) = M \quad (l(N) \cup l(R)) = \varnothing \quad l(\pi_M) = manage \quad l'(\pi_M) = wait}{(l, v) \xrightarrow{fm} (l', v')}$$

Table 13: Action fm (DUTA).

and the only nominal edge that is neither a termination nor a pause edge in the TTD counterpart, they are therefore *execute B* actions eb. Now, the edge from $main_{B\,pause}$ to $main_B$ does not match the definition of any action and will be thus treated as an internal action τ. Fig. 23 shows the DUTA of activity B where edges are tagged with their corresponding actions (non-clock guards and operations are omitted for readability).

$$ia.1: \frac{\exists ether \xrightarrow{ia} stop \quad l(\Pi) = ID_{A\in\mathcal{A}} \quad l(\pi_A) = ether \quad ID_A \in l(R) \quad l'(\pi_A) = stop \quad ID_A \notin l'(R) \quad v'(x_A) = 0}{(l,v) \xrightarrow{ia} (l',v')}$$

$$ia.2: \frac{\exists c_{pause} \xrightarrow{ia} stop \quad l(\Pi) = ID_{A\in\mathcal{A}} \quad l(\pi_A) = c_{pause} \quad ID_A \in l(R) \quad l'(\pi_A) = stop \quad ID_A \notin l'(R) \quad v'(x_A) = 0}{(l,v) \xrightarrow{ia} (l',v')}$$

$$ia.3: \frac{\exists ether \xrightarrow{ia} ether \quad l(\Pi) = ID_{A\in\mathcal{A}} \quad l(\pi_A) = ether \quad ID_A \in l(R) \quad l'(\pi_A) = ether \quad l'(\Pi) = M \quad \neg(ID_A \in l'(N) \vee ID_A \in l'(R))}{(l,v) \xrightarrow{ia} (l',v')}$$

$$ia.4: \frac{\exists c_{pause} \xrightarrow{ia} ether \quad l(\Pi) = ID_{A\in\mathcal{A}} \quad l(\pi_A) = c_{pause} \quad ID_A \in l(R) \quad l'(\pi_A) = ether \quad l'(\Pi) = M \quad \neg(ID_A \in l'(N) \vee ID_A \in l'(R))}{(l,v) \xrightarrow{ia} (l',v')}$$

Table 14: Actions ia (DUTA).

$$fa.1: \frac{\exists c \xrightarrow{fa} c'_{pause} \quad l(\Pi) = ID_{A\in\mathcal{A}} \quad ID_A \notin l(R) \quad l(\pi_A) = c \quad v(x_A) > 0 \quad l'(\pi_A) = c'_{pause} \quad l'(\Pi) = M \quad \neg(ID_A \in l'(N) \vee ID_A \in l'(R))}{(l,v) \xrightarrow{fa} (l',v')}$$

$$fa.2: \frac{\exists c \xrightarrow{fa} ether \quad l(\Pi) = ID_{A\in\mathcal{A}} \quad l(\pi_A) = c \quad ID_A \notin l(R) \quad v(x_A) > 0 \quad l'(\pi_A) = ether \quad l'(\Pi) = M \quad \neg(ID_A \in l'(N) \vee ID_A \in l'(R))}{(l,v) \xrightarrow{fa} (l',v')}$$

Table 15: Actions fa (DUTA).

The internal discrete action τ The internal action τ is possible only when the activity has the control, its current location is c_{pause} and it is not interrupted. Taking τ changes the current location to c and resets the clock of the activity (**Definition 17**). Formally, the inference rules are given in table 17.

Time actions (DUTA) The preconditions and effects of time actions are given by the inference rules in table 18 (X is the set of clocks in the DUTA system).

$$ea.1: \frac{l(\Pi) = ID_{A \in \mathcal{A}} \quad ID_A \notin l(R) \quad l(\pi_A) = ether \quad l'(\pi_A) = start \quad v'(x_A) = 0}{(l, v) \xrightarrow{ea} (l', v')}$$

$$ea.2: \frac{\exists c \neq ether \xrightarrow{ea} c' \quad l(\Pi) = ID_{A \in \mathcal{A}} \quad ID_A \notin l(R) \quad l(\pi_A) = c \quad v(x_A) > 0 \quad l'(\pi_A) = c' \quad v(x_A) = 0}{(l, v) \xrightarrow{ea} (l', v')}$$

Table 16: Actions ea (DUTA).

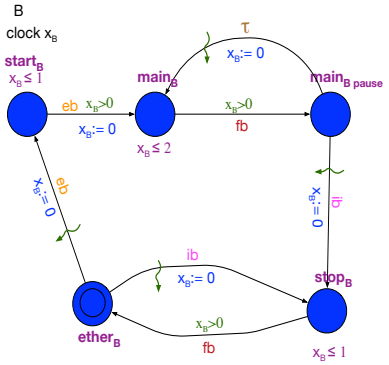


Fig. 23: Actions in DUTA of activity B

$$\tau: \frac{l(\pi_A) = c_{pause} \quad l(\Pi) = ID_{A \in \mathcal{A}} \quad ID_A \notin l(R) \quad l'(\pi_A) = c \quad v'(x_A) = 0}{(l, v) \xrightarrow{\tau} (l', v')}$$

Table 17: Internal action tau (DUTA).

6.2.2 Absence of st effect on activities The action st has no effect on the enabledness or timing constraints of activities actions.

Proof. The action st changes only the variable sig, that is not involved in any guard $g(act) \mid act \in \{ea, fa, ia\}$. It follows that if act is enabled (or disabled) before taking st, it will remain so after taking it, both in the TTDs and DUTA. As for timing constraints, Since the enabledness is not affected in the TTD then we may write:
$(s, \phi) \xrightarrow{st} (s', \phi') \Leftrightarrow \phi'(act) = \phi(act) \forall act \in \{ea, fa, ia\}$. In the DUTA, the clocks valuations are trivially unaffected because the timer has no access to the activities clocks, intrinsically local to their components.

6.2.3 Absence of external actions effects on timer Any action that is external to the timer has no effect on the enabledness or timing constraints of the timer action.

$$d.1: \frac{\begin{array}{c} v(xt) < Per \quad l(sig) = false \quad l(\pi_M) = wait \\ d \in]0, Per - v(xt)] \\ \forall x \in X : v'(x) = v(x) + d \end{array}}{(s, \phi) \xrightarrow{d} (s', \phi')}$$

$$d.2: \frac{\begin{array}{c} v(xt) < Per \quad l(\Pi) = ID_{A \in \mathcal{A}} \quad l(\pi_A) \neq ether \\ ID_A \notin l(R) \qquad v(x_A) < W(c) \\ d \in]0, min(Per - v(xt), W(c) - v(x_A))] \\ \forall x \in X : v'(x) = v(x) + d \end{array}}{(s, \phi) \xrightarrow{d} (s', \phi')}$$

Table 18: Time action d (DUTA).

Proof. The action st is guard free, which means that it is always enabled, and thus no other action can affect its enabledness or timing constraints. In the DUTA, the clock valuations is trivially unaffected because external actions have no access to the timer clock, intrinsically local to it.

6.2.4 Edges equivalence Let At and Ad be, respectively, the TTD and DUTA of some activity A. (i) there is an action edge ea between vertices c and c' in At iff there is an identical action edge between locations c and c' in Ad. (i) there is an action edge ia between vertices $ether$ and c' in At iff there is an identical action edge between locations $ether$ and c' in Ad. (iii) there is an action edge ia (respect. fa) between vertices c and c' in At iff there is an identical action edge between locations c_{pause} and c' (respect. c and c'_{pause}) in Ad, given that there is an edge $\rightarrow c \in E^P$ in At (respect. given that c$\rightarrow$ c' is in E^P). (iv) there is an action edge fa between vertices c and $ether$ in At iff there is an identical action edge between locations c and $ether$ in Ad. Formally:
(i) $\exists c_{At} \xrightarrow{ea} c'_{At} \Leftrightarrow \exists c_{Ad} \xrightarrow{ea} c'_{Ad}$
(ii) $\exists ether_{At} \xrightarrow{ia} c'_{At} \Leftrightarrow \exists ether_{Ad} \xrightarrow{ia} c'_{Ad}$
(iii) (action *ia*) $(\exists c_{At} \xrightarrow{ia} c'_{At} \wedge \exists \rightarrow c \in E^P) \Leftrightarrow \exists c_{_{Ad\ pause}} \xrightarrow{ia} c'_{Ad}$
(iii) (action *fa*) $(\exists c_{At} \xrightarrow{fa} c'_{At} \in E^P) \Leftrightarrow \exists c \xrightarrow{fa} c'_{_{Ad\ pause}}$
(iv) $\exists c_{At} \xrightarrow{fa} ether_{At} \Leftrightarrow \exists c_{Ad} \xrightarrow{fa} ether_{Ad}$

Proof. From **Definition 17**, each edge $c \rightarrow c'$ in the TTD is mapped to $c \rightarrow c'$ in the DUTA, except for interruption edges $c \rightarrow c'$ mapped to $c_{pause} \rightarrow c'$ iff c satisfies $\exists \rightarrow c \in E^P$ (respect. pause edges $c \rightarrow c'$ mapped to $c \rightarrow c'_{pause}$).

6.2.5 Bisimilarity between TTS and DUTA systems Let Ψ and Γ be the Kripke structures over which the semantics of an execution task TTS and DUTA, respectively, is defined. Each state in Ψ is a pair (s, ϕ) where s is the TTS state and ϕ the future dates for taking transitions (Sect. 4.1.3). Each state in Γ is a pair (l, v) where l is the interpretation of all variables excluding clocks and v the valuation of each clock x in the DUTA composition. The objective is to prove that Ψ and Γ are timed bisimilar.

Definition 20 ***Timed bisimilarity.***
We say that Ψ and Γ are timed bisimilar iff for some binary relation $\mathcal{R}$ and discrete or time-progress action α, the initial states of Ψ and Γ are in $\mathcal{R}$, that is $\psi_0 R \gamma_0$, and:

- *Γ simulates Ψ: if $(\psi \in \Psi) R (\gamma \in \Gamma)$ and $\psi \xrightarrow{\alpha} \psi'$ then $\exists \gamma' \in \Gamma$ s.t. $\gamma \xrightarrow{\alpha} \gamma'$ and $\psi' R \gamma'$,*
- *Ψ simulates Γ: if $(\psi \in \Psi) R (\gamma \in \Gamma)$ and $\gamma \xrightarrow{\alpha} \gamma'$ then $\exists \psi' \in \Psi$ s.t. $\psi \xrightarrow{\alpha} \psi'$ and $\psi' R \gamma'$.*

In our proof, we use the *weak* version of timed bisimilarity: some internal actions τ may be involved in $\xrightarrow{\alpha}$. That is, $x(\xrightarrow{\tau})^* \xrightarrow{\alpha} (\xrightarrow{\tau})^* x'$ is observed simply as $x \xrightarrow{\alpha} x'$. For a stronger equivalence between the models, we require τ to be a discrete action. That is, time is not allowed to progress when taking τ. We begin by defining the relation $\mathcal{R}$:

Definition 21 ***The binary relation $\mathcal{R}$:***

$$(\psi = (s, \phi)) \mathcal{R} (\gamma = (l, v)) \; iff \begin{cases} (1)\ (\forall u \in \{\Pi, N, R, sig\} : s(u) = l(u)) \wedge \\ \\ (2)\ (s(\pi_{Tim}) = l(\pi_{Tim}) \wedge \phi(st) = I_{st} - \theta \wedge v(xt) = \theta) \wedge \\ \\ (3)\ (s(\pi_M) = l(\pi_M)) \wedge \\ \\ (4)\ (\forall A \in \mathcal{A} \mid s(\Pi) = l(\Pi) \neq ID_A : s(\pi_A) = l(\pi_A) \vee \\ (s(\pi_A) = c \wedge l(\pi_A) = c_{pause})) \wedge \\ \\ (5)(\exists A \models (s(\Pi) = l(\Pi) = ID_A) \Rightarrow \\ (5.1)(s(\pi_A) = l(\pi_A) = ether) \vee \\ (5.2)(s(\pi_A) = l(\pi_A) = c \neq ether \wedge ((\phi(ea) = I_{ea} - \theta) \vee \\ \qquad (\phi(fa) = I_{fa} - \theta)) \wedge v(x_A) = \theta) \vee \\ (5.3)(s(\pi_A) = c \wedge l(\pi_A) = c_{pause} \wedge \\ \qquad (ID_A \in s(R) \vee ((\phi(ea) = I_{ea} \vee \\ \qquad \phi(fa) = I_{fa}))))) \end{cases}$$

Informally, **Definition 21** of the relation between states ψ and γ says the following. Rules (1) to (4) stipulate that ψ and γ need to agree on all state variables, at the exception of the locations of idle activities (not being executed) that can be c_{pause} instead of c in γ. Additionally, the property $\phi(st) = I_{st} - \theta \wedge v(xt) = \theta$ (Rule (2)) reflects that time in both timers progresses at the same rate (there is no guard on st which means $\phi(st)$ is always defined, and taking st in the timer DUTA resets xt). Rule (5) is only for the activity A currently executing (if any). Roughly, it says that the TTD vertex and the DUTA location of A need to be identical, and at which time must progress similarly. The location and vertex of the activity DUTA and TTD, respectively, must be identical if at $ether$ (5.1). The location of the DUTA of A must match the vertex of its TTD counterpart when executing a codel, where time must also progress identically (5.2). Finally, if the vertex of the TTD is c whereas the location of the DUTA is c_{pause}, then time is not allowed to progress and only instantaneous actions (mainly interruption actions) are possible (5.3).

Initial states We start with checking whether the initial states $\psi_0 = (s_0, \phi_0)$ and $\gamma_0 = (l_0, v_0)$ are in $\mathcal{R}$ (**Definition 20**). By definition, we know that initially:
$(s_0(N) = s_0(R) = l_0(N) = l_0(R) = \varnothing) \wedge (s_0(sig) = l_0(sig) = false) \wedge$
$(s_0(\Pi) = l_0(\Pi) = M)$,
$l_0(\pi_{Tim}) = s_0(\pi_{Tim}) = start \wedge \phi(st) = I_{st} \wedge v(xt) = 0$,
$l_0(\pi_M) = s_0(\pi_M) = wait$,
$\forall A \in \mathcal{A} : l_0(\pi_A) = s_0(\pi_A) = ether$,
$\nexists A \in \mathcal{A} \mid l_0(\Pi) = s_0(\Pi) = ID_A$

It follows that ψ_0 and γ_0 satisfy all the rules in **Definition 21**, that is $\psi_0 \mathcal{R} \gamma_0$.
Now, we prove that Γ (weakly) time simulates Ψ (**Definition 20**). Let $\psi \in \Psi$ and $\gamma \in \Gamma$ be some states satisfying $\psi \mathcal{R} \gamma$.

Discrete actions

Action st**:** From inference rules in table 2, to take st from ψ, we must have:
$\psi = (s, \phi) \models (0 \in \phi(st))$ (1.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (1.a) and **Definition 21** (Rule (2)) we have:
$v(xt) = \theta \wedge \phi(st) = I_{st} - \theta$, knowing that $I_{st} = [Per, Per]$ and from (1.a) $0 \in \phi(st)$.
It follows that $\theta = Per$ and thus $v(xt) = Per$ (1.b)
Now from inference rules in table 10, to take st we must have
$\gamma = (l, v) \models (v(xt) = Per)$ (1.c)
From (1.b) and (1.c) it follows that action st is possible at γ.
We take now the action st from ψ to reach the state ψ'. From table 2 we have:
$\psi' = (s', \phi') \models (s'(sig) = true \wedge \phi(st) = [Per, Per])$ and s' agrees with s otherwise (1.d)
We take the action st from γ to reach the state γ'. From table 10 we have:
$\gamma' = (l', v') \models (l'(sig) = true \wedge v'(xt) = 0)$ and l' agrees with l otherwise (1.e)
From (1.d) and (1.e) it follows that rules (1) to (4) in **Definition 21** are satisfied by ψ' and γ', and from Sect. 6.2.2 (absence of effects on activities) we conclude that the rule (5) in **Definition 21** is satisfied as well.
It follows that $\psi' \mathcal{R} \gamma'$.

Action sm**:** From inference rules in table 3, to take sm from ψ, we must have:
$\psi = (s, \phi) \models (s(sig) = true \wedge s(\pi_M) = wait)$ (2.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (2.a) and **Definition 21** (Rules (1),(3)) we have at γ:
$l(sig) = true \wedge l(\pi_M) = wait$ (2.b)
Now from inference rules in table 11, to take sm we must have
$\gamma = (l, v) \models (l(sig) = true \wedge l(\pi_M) = wait)$ (2.c)
From (2.b) and (2.c) it follows that action sm is possible at γ.
We take now the action sm from ψ to reach the state ψ'. From table 3 we have:
$\psi' = (s', \phi') \models (s'(sig) = false \wedge s'(\pi_M) = manage \wedge s'(N) = N' \wedge s'(R) = R')$ and s' agrees with s otherwise (2.d)
We take the action sm from γ to reach the state γ'. From table 11 we have:
$\gamma' = (l', v') \models (l'(sig) = false \wedge l'(\pi_M) = manage \wedge l'(N) = N' \wedge l'(R) = R')$ and l' agrees with l otherwise (2.e)

From (2.d) and (2.e) and absence of external actions effect on the timer (Sect. 6.2.3) it follows that rules (1) to (4) in **Definition 21** are satisfied by ψ' and γ'. Also, from (2.a) and (2.d) (respect. (2.b) and (2.e)) we have $\Pi = M$ at ψ' and γ' and thus both satisfy rule (5) in **Definition 21** ($\nexists A \in \mathcal{A} \mid s'(\Pi) = l'(\Pi) = ID_A$).
It follows that $\psi' \mathcal{R} \gamma'$.

Note that N' and R', being the result of a random initialization, may be different at ψ' and γ'. However, since the operation $rrand(N, R)$ is the same in both systems, it is sufficient to match the states pairwise, that is ψ' and γ' where the result of applying $rrand(N, R)$ is the same. It is trivial to prove that mapping $\psi' \in \Psi$ to $\gamma' \in \Gamma$ s.t. $\xrightarrow{sm} \psi' \wedge \xrightarrow{sm} \gamma' \wedge \psi(N') = \gamma(N') \wedge \psi(R') = \gamma(R')$ is a one-to-one function defined over all $\psi' \models \xrightarrow{sm} \psi'$, and thus for each ψ' resulting from taking an action sm there is $\gamma' \in \Gamma$ s.t. $\psi' \mathcal{R} \gamma'$.

Action lm**:** From inference rules in table 4, to take lm from ψ, we must have:
$\psi = (s, \phi) \models (s(\Pi) = M \wedge (s(N) \cup s(R)) \neq \varnothing)$ (3.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (3.a) and **Definition 21** (Rule (1)) we have at γ:
$l(\Pi) = M \wedge (l(N) \cup l(R)) \neq \varnothing$ (3.b)
Now from inference rules in table 12, to take lm we must have
$\gamma = (l, v) \models (l(\Pi) = M \wedge (l(N) \cup l(R)) \neq \varnothing)$ (3.c)
From (3.b) and (3.c) it follows that action lm is possible at γ.
We take now the action lm (rule $lm.1$) from ψ to reach the state ψ'. From table 4 we have:
$\psi' = (s', \phi') \models (s'(\Pi) = ID_{A \in \mathcal{A}} \wedge s'(\pi_A) = ether \wedge ID_A \in s'(R))$ and s' agrees with s otherwise (3.d)
We take the action lm (rule $lm.1$) from γ to reach the state γ'. From table 12 we have:
$\gamma' = (l', v') \models (l'(\Pi) = ID_{A \in \mathcal{A}} \wedge l'(\pi_A) = ether \wedge ID_A \in l'(R))$ and l' agrees with l otherwise (3.e)
From (3.d) and (3.e) and absence of external actions effect on the timer (Sect. 6.2.3) it follows that rules (1) to (4) as well as rule (5) (through the clause 5.1) are satisfied by ψ' and γ', that is $\psi' \mathcal{R} \gamma'$ after taking lm (rule $lm.1$).

We take now the action lm (rule $lm.2$) from ψ to reach the state ψ'. From table 4 we have:
$\psi' = (s', \phi') \models (s'(\Pi) = ID_{A \in \mathcal{A}} \wedge s'(\pi_A) = c \neq ether \wedge ID_A \in s'(R))$ and s' agrees with s otherwise (3.f)
We take the action lm (rule $lm.2$) from γ to reach the state γ'. From table 12 we have:
$(l', v') \models (l'(\Pi) = ID_{A \in \mathcal{A}} \wedge l'(\pi_A) = c_{pause} \wedge ID_A \in l'(R))$ and l' agrees with l otherwise (3.g)
From (3.f) and (3.g) and absence of external actions effect on the timer (Sect. 6.2.3) it follows that rules (1) to (4) as well as rule (5) (through the clause 5.3) are satisfied by ψ' and γ', that is $\psi' \mathcal{R} \gamma'$ after taking lm (rule $lm.2$).

We take now the action lm (rule $lm.3$) from ψ to reach the state ψ'. From table 4 we have:
$\psi' = (s', \phi') \models (s'(\Pi) = ID_{A \in \mathcal{A}} \wedge s'(\pi_A) = ether \wedge ID_A \notin s'(R))$ and s' agrees with s otherwise (3.h)

We take the action lm (rule $lm.3$) from γ to reach the state γ'. From table 12 we have:
$\gamma' = (l', v') \models (l'(\Pi) = ID_{A \in \mathcal{A}} \wedge l'(\pi_A) = ether \wedge ID_A \notin l'(R))$ and l' agrees with l otherwise (3.i)
From (3.h) and (3.i) and absence of external actions effect on the timer (Sect. 6.2.3) it follows that rules (1) to (4) as well as rule (5) (through the clause 5.1) are satisfied by ψ' and γ', that is $\psi' \mathcal{R} \gamma'$ after taking lm (rule $lm.3$).

We take now the action lm (rule $lm.4$) from ψ to reach the state ψ'. From table 4 we have:
$\psi' = (s', \phi') \models (s'(\Pi) = ID_{A \in \mathcal{A}} \wedge s'(\pi_A) = c \neq ether \wedge ID_A \notin s'(R) \wedge$
$(\phi'(ea) = I_{ea} \vee \phi'(fa) = I_{fa}))$ and s' agrees with s otherwise (3.j)
We take the action lm (rule $lm.4$) from γ to reach the state γ''. From table 12 we have:
$\gamma'' = (l'', v'') \models (l''(\Pi) = ID_{A \in \mathcal{A}} \wedge l''(\pi_A) = c_{pause} \wedge ID_A \notin l''(R))$ and l'' agrees with l otherwise.
We take now the internal urgent action τ from γ'' to reach the state γ'. From table 17 we have:
$\gamma' = (l', v') \models (l'(\Pi) = l'(\pi_A) = c \wedge v'(x_A) = 0)$ and l' agrees with l'' otherwise (3.k)
From (3.j) and (3.k) and absence of external actions effect on the timer (Sect. 6.2.3) it follows that rules (1) to (4) as well as rule (5) (through the clause 5.2[6] with $\theta = 0$) are satisfied by ψ' and γ', that is $\psi' \mathcal{R} \gamma'$ after taking lm (rule $lm.4$).

Action fm**:** From inference rules in table 5, to take fm from ψ, we must have:
$\psi = (s, \phi) \models (s(\Pi) = M \wedge (s(N) \cup s(R)) = \varnothing \wedge s(\pi_M) = manage)$ (4.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (4.a) and **Definition 21** (Rules (1),(3)) we have at γ:
$l(\Pi) = M \wedge (l(N) \cup l(R)) = \varnothing \wedge l(\pi_M) = manage$ (4.b)
Now from inference rules in table 13, to take fm we must have
$\gamma = (l, v) \models (l(\Pi) = M \wedge (l(N) \cup l(R)) = \varnothing \wedge l(\pi_M) = manage)$ (4.c)
From (4.b) and (4.c) it follows that action fm is possible at γ.
We take now the action fm from ψ to reach the state ψ'. From table 5 we have:
$\psi' = (s', \phi') \models (s'(\pi_M) = wait)$ and s' agrees with s otherwise (4.d)
We take the action fm from γ to reach the state γ'. From table 13 we have:
$\gamma' = (l', v') \models (l'(\pi_M) = wait)$ and l' agrees with l otherwise (4.e)
From (4.d) and (4.e) and absence of external actions effect on the timer (Sect. 6.2.3) it follows that rules (1) to (4) in **Definition 21** are satisfied by ψ' and γ'. Also, from (4.a) and (4.d) (respect. (4.b) and (4.e)) we have $\Pi = M$ at both ψ' and γ' and thus both satisfy rule (5) in **Definition 21** ($\nexists A \in \mathcal{A} \mid s'(\Pi) = l'(\Pi) = ID_A$).
It follows that $\psi' \mathcal{R} \gamma'$.

Action ia**:** From inference rules in table 6, to take ia (rule $ia.1$ or $ia.3$) from ψ, we must have, besides the existence of an outgoing ia edge from $ether$ (to $stop$ (rule $ia.1$) or to $ether$ ($ia.3$)):
$\psi = (s, \phi) \models (s(\Pi) = ID_{A \in \mathcal{A}} \wedge s(\pi_A) = ether \wedge ID_A \in s(R))$ (5.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (5.a) and **Definition 21** (Rules (1), (5.1))

[6] Note that we know that location c from (3.k) is different from $ether$ since c_{pause} exists and we know by definition (Sect. 5.2.5) that $ether$ cannot be the target of a pause.

we have at γ:
$l(\Pi) = ID_{A \in \mathcal{A}} \wedge l(\pi_A) = ether \wedge ID_A \in l(R)$ (5.b)
Now from inference rules in table 14, to take ia (rule $ia.1$ or $ia.3$) we must have, besides the existence of an outgoing ia edge from $ether$ (to $stop$ (rule $ia.1$) or to $ether$ ($ia.3$)):
$\gamma = (l, v) \models (l(\Pi) = ID_{A \in \mathcal{A}} \wedge l(\pi_A) = ether \wedge ID_A \in l(R))$ (5.c)
From (5.a), (5.b) and (5.c) and edges equivalence (Sect. 6.2.4) it follows that action ia (rule $ia.1$ or $ia.3$) is possible at γ.
We take now the action ia (rule $ia.1$) from ψ to reach the state ψ'. From table 6 we have:
$\psi' = (s', \phi') \models (s'(\pi_A) = stop \wedge ID_A \notin s'(R) \wedge (\phi' ea = I_{ea} \vee \phi'(fa) = I_{fa}))$ and s' agrees with s otherwise (5.d).
We take the action ia (rule $ia.1$) from γ to reach the state γ'. From table 14 we have:
$\gamma' = (l', v') \models (l'(\pi_A) = stop \wedge v'(x_A) = 0 \wedge ID_A \notin l'(R))$ and l' agrees with l otherwise (5.e)
From (5.d) and (5.e) and absence of external actions effect on the timer (Sect. 6.2.3) it follows that rules (1) to (4) as well as rule (5) (through the clause 5.2 with $\theta = 0$) are satisfied by ψ' and γ', that is $\psi' \mathcal{R} \gamma'$ after taking ia (rule $ia.1$).

We take now the action ia (rule $ia.3$) from ψ to reach the state ψ'. From table 6 we have:
$\psi' = (s', \phi') \models (s'(\pi_A) = ether \wedge s'(\Pi) = M \wedge \neg(ID_A \in s'(N) \vee ID_A \in s'(R)))$ and s' agrees with s otherwise (5.f)
We take the action ia (rule $ia.3$) from γ to reach the state γ'. From table 14 we have:
$\gamma' = (l', v') \models (l'(\pi_A) = ether \wedge l'(\Pi) = M \wedge \neg(ID_A \in l'(N) \vee ID_A \in l'(R)))$ and l' agrees with l otherwise (5.g)
From (5.f) and (5.g) and absence of external actions effect on the timer (Sect. 6.2.3) it follows that rules (1) to (5) are satisfied by ψ' and γ' ((5) is satisfied because $\Pi = M$ at both ψ' and γ' and thus $\nexists A \in \mathcal{A} \mid s'(\Pi) = l'(\Pi) = ID_A$). It follows that $\psi' \mathcal{R} \gamma'$ after taking $ia.1$ or $ia.3$.

From inference rules in table 6, to take ia (rule $ia.2$ or $ia.4$) from ψ, we must have, besides the existence of an outgoing ia edge from c (to $stop$ (rule $ia.2$) or to $ether$ ($ia.4$)):
$\psi = (s, \phi) \models (s(\Pi) = ID_{A \in \mathcal{A}} \wedge s(\pi_A) = c \neq ether \wedge ID_A \in s(R))$ (5.h)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (5.h) and **Definition 21** (Rules (1),(5.3)) we have at γ:
$l(\Pi) = ID_{A \in \mathcal{A}} \wedge l(\pi_A) = c_{pause} \wedge ID_A \in l(R)$ (5.i)
Now from table 14, to take ia (rule $ia.2$ or $ia.4$) we must have, besides the existence of an outgoing ia edge from c_{pause} (to $stop$ (rule $ia.2$) or to $ether$ ($ia.4$)):
$\gamma = (l, v) \models (l(\Pi) = ID_{A \in \mathcal{A}} \wedge l(\pi_A) = c_{pause} \wedge ID_A \in l(R)$ (5.j)
From (5.h), (5.i), (5.j) and edges equivalence (Sect. 6.2.4) it follows that action ia (rule $ia.2$ or $ia.4$) is possible at γ.
Now, proving that $\psi' \mathcal{R} \gamma'$ after applying rule $ia.2$ (respect. $ia.4$) is identical to proving $\psi' \mathcal{R} \gamma'$ after applying rule $ia.1$ (respect. $ia.3$).

Action fa**:** From inference rules in table 7, to take fa from ψ, we must have, besides the existence of an outgoing fa edge from c (to $c' \neq ether$ (rule $fa.1$) or to $ether$ ($fa.2$)):

$\psi = (s, \phi) \models (s(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin s(R) \wedge s(\pi_A) = c \wedge (\phi(fa) = I_{fa} - \theta \mid \theta > 0))$ (6.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (6.a) and **Definition 21** (Rules (1),(5.2)[7]) we have at γ:
$l(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin l(R) \wedge l(\pi_A) = c \wedge v(x_A) > 0$ (6.b)
Now from From inference rules in table 15, to take fa we must have, besides the existence of an outgoing fa edge from c (to c'_{pause} (rule $fa.1$) or to $ether$ ($fa.2$)):
$\gamma = (l, v) \models (l(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin l(R) \wedge l(\pi_A) = c \wedge v(x_A) > 0)$ (6.c)
From (6.a), (6.b), (6.c) and edges equivalence (Sect. 6.2.4) it follows that action fa is possible at γ.
We take now the action fa (rule $fa.1$) from ψ to reach the state ψ'. From table 7 we have:
$\psi' = (s', \phi') \models (s'(\pi_A) = c' \neq ether \wedge s'(\Pi) = M \wedge \neg(ID_A \in s'(N) \vee ID_A \in s'(R)))$
and s' agrees with s otherwise (6.d)
We take now the action fa (rule $fa.1$) from γ to reach the state γ'. From table 15 we have:
$\gamma' = (l', v') \models (l'(\pi_A) = c'_{pause}, l'(\Pi) = M \wedge \neg(ID_A \in l'(N) \vee ID_A \in l'(R)))$ and l' agrees with l otherwise (6.e)
From (6.d) and (6.e) and absence of external actions effect on the timer (Sect. 6.2.3) it follows that rules (1) to (4) in **Definition 21** are satisfied by ψ' and γ'. Also, we have $\Pi = M$ at both ψ' and γ' and thus both satisfy rule (5) in **Definition 21** ($\nexists A \in \mathcal{A} \mid s'(\Pi) = l'(\Pi) = ID_A$).
It follows that $\psi' \mathcal{R} \gamma'$.

We take now the action fa (rule $fa.2$) from ψ to reach the state ψ'. From table 7 we have:
$\psi' = (s', \phi') \models (s'(\pi_A) = ether \wedge s'(\Pi) = M \wedge \neg(ID_A \in s'(N) \vee ID_A \in s'(R)))$ and s' agrees with s otherwise (6.f)
We take now the action fa (rule $fa.2$) from γ to reach the state γ'. From table 15 we have:
$\gamma' = (l', v') \models (l'(\pi_A) = ether \wedge l'(\Pi) = M \wedge \neg(ID_A \in l'(N) \vee ID_A \in l'(R)))$ and l' agrees with l otherwise (6.g)
From (6.f) and (6.g) and absence of external actions effect on the timer (Sect. 6.2.3) it follows that rules (1) to (4) in **Definition 21** are satisfied by ψ' and γ'. Also, from (6.a) and (6.f) (respect. (6.b) and (6.g)) we have $\Pi = M$ at both ψ' and γ' and thus both satisfy rule (5) in **Definition 21** ($\nexists A \in \mathcal{A} \mid s'(\Pi) = l'(\Pi) = ID_A$).
It follows that $\psi' \mathcal{R} \gamma'$.

Action ea**:** From inference rules in table 8, to take ea (rule $ea.1$) from ψ, we must have:
$\psi = (s, \phi) \models (s(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin s(R) \wedge s(\pi_A) = ether)$ (7.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (7.a) and **Definition 21** (Rules (1),(5.1)) we have at γ:
$l(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin l(R) \wedge l(\pi_A) = ether$ (7.b)
Now from inference rules in table 16, to take ea (rule $ea.1$) we must have:

[7] Here also we know that c in (6.a) is different from $ether$ (Sect. 5.2.5 and **Definition 1**, there is no nominal edge outgoing $ether$).

$\gamma = (l, v) \models (l(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin l(R) \wedge l(\pi_A) = ether)$ (7.c)
From (7.b), (7.c) it follows that action ea is possible at γ.
We take now the action ea (rule $ea.1$) from ψ to reach the state ψ'. From table 8 we have:
$\psi' = (s', \phi') \models (s'(\pi_A) = start \wedge (\phi'(ea) = I_{ea} \vee \phi'(fa) = I_{fa}))$ and s' agrees with s otherwise (7.d)
We take the action ea (rule $ea.1$) from γ to reach the state γ'. From table 16 we have:
$\gamma' = (l', v') \models (l'(\pi_A) = start \wedge v'(x_A) = 0)$ and l' agrees with l otherwise (7.e)
From (7.d) and (7.e) and absence of external actions effect on the timer (Sect. 6.2.3) it follows that rules (1) to (4) as well as rule (5) (through the clause 5.2 with $\theta = 0$) are satisfied by ψ' and γ'.
It follows that $\psi' \mathcal{R} \gamma'$.

From inference rules in table 8, to take ea (rule $ea.2$) from ψ, we must have, besides the existence of an outgoing ea edge from c:
$\psi = (s, \phi) \models (s(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin s(R) \wedge s(\pi_A) = c \neq ether \wedge \phi(ea) = I_{ea} - \theta \mid \theta > 0)$ (7.f)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (7.f) and **Definition 21** (Rules (1),(5.2)) we have at γ:
$l(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin l(R) \wedge l(\pi_A) = c \neq ether \wedge v(x_A) > 0$ (7.g)
Now from inference rules in table 16, to take ea (rule $ea.2$) we must have, besides the existence of an outgoing ea edge from c:
$\gamma = (l, v) \models (l(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin l(R) \wedge l(\pi_A) = c \neq ether \wedge v(x_A) > 0)$ (7.h)
From (7.f), (7.g), (7.h) and edges equivalence (Sect. 6.2.4) it follows that action ea (rule $ea.2$) is possible at γ.
We apply now the rule $ea.2$ from ψ to reach the state ψ' (table 8) then from γ to reach the state γ' (table 16). The proof that $\psi' \mathcal{R} \gamma'$ is similar to that when taking $ea.1$ (with replacing $start$ by c'.

Time actions From inference rules in table 9, to take d (rule $d.1$) from ψ, we must have:
$\psi = (s, \phi) \models ((\phi(st) = I_{st} - a \mid a < Per) \wedge s(sig) = false \wedge s(\pi_M) = wait)$ (8.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (8.a) and **Definition 21** (Rules (1),(2, $\theta = a$),(3)) we have at γ:
$v(xt) = a \wedge l(sig) = false \wedge l(\pi_M) = wait$ (8.b)
Now from inference rules in table 18, to take d we must have:
$(l, v) \models (v(xt) < Per \wedge l(sig) = false \wedge l(\pi_M) = wait)$ (8.c)
From (8.a), (8.b) and (8.c) it follows that action d ($d \in]0, Per - a]$) is possible at γ.
We take now the action d ($d \in]0, Per - a]$) from ψ to reach the state ψ'. From table 9 (rule $d.1$) we have:
$\psi' = (s, \phi') \models (\phi'(st) = I_{st} - a - d)$ (8.d).
We take the action d ($d \in]0, Per - a]$) from γ to reach the state γ'. From table 18 (rule $d.1$) we have:
$\gamma' = (l, v') \models (\forall x \in X : v'(x) = v(x) + d)$, which means $v'(xt) = a + d$ (8.e)
From (8.d) and (8.e) it follows that rules (1), (2 with $\theta = a + d$), (3), (4), and (5) (the manager has the control because $s(\pi_M) = l(\pi_M) = wait$ and thus $\nexists A \in \mathcal{A} \mid s(A) = l(A) = ID_A$) are satisfied by ψ' and γ'.

It follows that $\psi' \mathcal{R} \gamma'$.

From inference rules in table 9, to take d (rule $d.2$) from ψ, we must have:
$\psi = (s, \phi) \models ((\phi(st) = I_{st} - a \mid a < Per) \wedge s(\Pi) = ID_A \wedge$
$s(\pi_A) = c \neq ether \wedge ID_A \notin s(R) \wedge (\phi(ea) = I_{ea} - b \vee \phi(fa) = I_{fa} - b \mid b < W(c)))$
(8.f)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (8.a) and **Definition 21** (Rules (1),(2, $\theta = a$),(5.2, $\theta = b$)) we have at γ:
$v(xt) = a \wedge l(\Pi) = ID_A \wedge$
$l(\pi_A) = c \wedge ID_A \notin l(R) \wedge v(x_A) = b$ (8.g)
Now from inference rules in table 18 (rule $d.2$), to take d we must have:
$(l, v) \models (v(xt) < Per \wedge l(\Pi) = ID_A \wedge$
$l(\pi_A) = c \neq ether \wedge ID_A \notin R \wedge v(x_A) < W(c))$ (8.h)
From (8.f), (8.g) and (8.h) it follows that action d ($d \in]0, min(W(c) - b, Per - a)]$) is possible at γ.
We take now the action d ($d \in]0, min(W(c) - b, Per - a)]$) from ψ to reach the state ψ'. From table 9 (rule $d.2$) we have:
$\psi' = (s, \phi') \models (\phi'(st) = I_{st} - a - d \wedge (\phi'(ea) = I_{ea} - b - d \vee \phi'(fa) = I_{fa} - b - d))$
(8.i).
We take the action d ($d \in]0, min(W(c) - b, Per - a)]$) from γ to reach the state γ'.
From table 18 (rule $d.2$) we have:
$\gamma' = (l, v') \models (\forall x \in X : v'(x) = v(x) + d)$, which means $(v'(xt) = a + d \wedge v'(x_A) = b + d)$
(8.j)
From (8.i) and (8.j) it follows that rules (1), (2 with $\theta = a + d$), (3), (4), and (5) (through the clause 5.2 with $\theta = b + d$) are satisfied by ψ' and γ'.
It follows that $\psi' \mathcal{R} \gamma'$.

We have thus proven that for all discrete and time actions, if $\psi \mathcal{R} \gamma$ and action act is possible from ψ s.t. $\psi \xrightarrow{act} \psi'$, then the same action act is possible from γ s.t. $\gamma \xrightarrow{act} \gamma'$ and $\psi' \mathcal{R} \gamma'$. It follows that Γ **(weakly) time simulates** Ψ. Similarly, we prove that Ψ **simulates** Γ in Appendix. A. Γ and Ψ are thus weakly timed bisimilar which proves the soundness of our translation.

7 Conclusion

In this document, we formalize a lightweight version of GEMOMF. The formal definitions and semantics give an unambiguous characterization of the most complex GEMOMF constituents, namely activities, execution and control tasks. We provide a sort of abstract syntax that, despite helping practitioners grasp the notion of components and their ingredients, defines the attributes on which operational semantics are built. That is, each abstract element in a tuple has an operational meaning that helps define the behavior of the global system. The work on semanticizing GEMOMF allowed to clarify several ambiguous notions such as the incompatibility between activities and the behavior of pause transitions. Additionally, it allowed, using the full power of TTS, GEMOMF to evolve from a single-threaded version, where tasks executed sequentially using a global

lock, to a multithreaded one where tasks run in parallel following a fine-grain mutual exclusion model.

In contrast to the descriptions given in Sect. 2, the operational semantics favors unambiguity and gives a clear view on the behavior of GenoM3 components. Indeed, the semantics given in this document in terms of TTDs composed in parallel would always give the same TTS for the same GenoM3 specifications, while informal descriptions might be interpreted in different ways. Also, besides the choice of TTS, only elementary operations over sets and booleans are used which abstracts away from more tedious structures and complex operators and contributes thus to the comprehension of the formalization. This smoothes translating the semantics to other formalisms and proving the soundness of such translations.

We also develop a sound translation from the TTS semantics to DUTA. We thus have, at the end of this document, accurate semantics of GenoM3 in the underlying formalisms of several formal frameworks. Indeed, the TTS semantics allowed the automatic generation from GenoM3 to Fiacre, presented in [13], as Fiacre specifications are a particular implementation of TTS. Moreover, the translation presented here enables mapping GenoM3 into pioneer frameworks based on DUTA and their subclasses, such as UPPAAL and BIP. This adds to the value and the usability of the work presented in this document.

References

[1] Tesnim Abdellatif, Saddek Bensalem, Jacques Combaz, Lavindra De Silva, and Félix Ingrand. Rigorous design of robot software: A formal component-based approach. *Robotics and Autonomous Systems*, 60(12):1563–1578, 2012.

[2] Rachid Alami, Raja Chatila, Sara Fleury, Malik Ghallab, and Félix Ingrand. An architecture for autonomy. *The International Journal of Robotics Research*, 17(4):315–337, 1998.

[3] Rajeev Alur and David Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[4] Ananda Basu, Bensalem Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous component-based system design using the bip framework. *IEEE software*, 28(3):41–48, 2011.

[5] Gerd Behrmann, Alexandre David, and Kim Larsen. A tutorial on uppaal. In *Formal Methods for the Design of Real-Time Systems*, pages 200–236. Springer, 2004.

[6] Albert Benveniste and Gérard Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79:1270–1282, 1991.

[7] Bernard Berthomieu, Jean-Paul Bodeveix, Patrick Farail, Mamoun Filali, Hubert Garavel, Pierre Gaufillet, Frederic Lang, and François Vernadat. Fiacre: an intermediate language for model verification in the topcased environment. In *European Congress on Embedded Real-Time Software and Systems*, 2008.

[8] Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis. Modeling urgency in timed systems. In *International Symposium on Compositionality: the significant difference*, pages 103–129. Springer, 1998.

[9] Frédéric Boussinot and Robert de Simone. The ESTEREL Language. In *Proceeding of the IEEE*, volume 79, pages 1293–1304, 1991.

[10] Herman Bruyninckx. Open robot control software: the OROCOS project. In *International Conference on Robotics and Automation*, pages 2523–2528. IEEE, 2001.

[11] M. Foughali. Toward a correct-and-scalable verification of concurrent robotic systems: Insights on formalisms and tools. In *IEEE Application of Concurrency to System Design*, pages 29–38, 2017.

[12] Mohammed Foughali, Bernard Berthomieu, Silvano Dal Zilio, Pierre-Emmanuel Hladik, Félix Ingrand, and Anthony Mallet. Formal verification of complex robotic systems on resource-constrained platforms. In *International Conference on Formal Methods in Software Engineering*, pages 2–9, 2018.

[13] Mohammed Foughali, Bernard Berthomieu, Silvano Dal Zilio, Félix Ingrand, and Anthony Mallet. Model checking real-time properties on the functional layer of autonomous robots. In *International Conference on Formal Engineering Methods*, pages 383–399. Springer, 2016.

[14] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and Andrew Roscoe. Fdr3a modern refinement checker for csp. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 187–201. Springer, 2014.

[15] Raju Halder, José Proença, Nuno Macedo, and André Santos. Formal verification of ros-based robotic applications using timed-automata. In *International Workshop on Formal Methods in Software Engineering (FormaliSE)*, pages 44–50. IEEE/ACM, 2017.

[16] Mohammed Hazim, Hongyang Qu, and Sandor Veres. Testing, verification and improvements of timeliness in ros processes. In *Conference Towards Autonomous Robotic Systems*, pages 146–157. Springer, 2016.

[17] Thomas Henzinger, Zohar Manna, and Amir Pnueli. Timed transition systems. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*, pages 226–251. Springer, 1991.

[18] Thomas Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and computation*, 111(2):193–244, 1994.

[19] Félix Ingrand and Malik Ghallab. Deliberation for autonomous robots: A survey. *Artificial Intelligence*, 247:10–44, 2017.

[20] Félix Ingrand, Simon Lacroix, Solange Lemai-Chenevier, and Frederic Py. Decisional autonomy of planetary rovers. *Journal of Field Robotics*, 24(7):559–580, 2007.

[21] Moonzoo Kim and Kyo Kang. Formal Construction and Verification of Home Service Robots: A Case Study. In *International Symposium on Automated Technology for Verification and Analysis*, pages 429–443. Springer, 2005.

[22] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. In *International Conference on Computer Aided Verification*, pages 585–591. Springer, 2011.

[23] Anthony Mallet, Cédric Pasteur, Mathieu Herrb, Séverin Lemaignan, and Félix Ingrand. GenoM3: Building middleware-independent robotic components. In *International Conference on Robotics and Automation*, pages 4627–4632. IEEE, 2010.

[24] Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, and Jon Timmis. Automatic property checking of robotic applications. In *International Conference on Intelligent Robots and Systems*, pages 3869–3876. IEEE, 2017.

[25] Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, Jon Timmis, and Jim Woodcock. Robochart: a state-machine notation for modelling and verification of mobile and autonomous robots. Technical report, University of York, 2016.

[26] Charles Pecheur. Verification and validation of autonomy software at nasa. Technical report, NASA Ames Research Center, 2000.

[27] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, page 5, 2009.

[28] Andrew Roscoe. *Understanding concurrent systems*. Springer Science & Business Media, 2010.

[29] Daniel Simon, Roger Pissard-Gibollet, and Soraya Arias. Orccad, a framework for safe robot control design and implementation. In *National workshop on control architectures of robots: software approaches and issues*, 2006.

[30] Arcot Sowmya, David Tsz-Wang So, and Wan Hung Tang. Design of a Mobile Robot Controller using Esterel Tools. *Electronic Notes in Theoretical Computer Science*, 65(5):3–10, 2002.

[31] Vassil Todorov, Frédéric Boulanger, and Safouan Taha. Formal verification of automotive embedded software. In *6th Conference on Formal Methods in Software Engineering*, pages 84–87. ACM, 2018.

[32] Richard Volpe, Issa Nesnas, Tara Estlin, Darren Mutz, Richard Petras, and Hari Das. The claraty architecture for robotic autonomy. In *Proccedings of IEEE Aerospace Conference*, pages 1–121, 2001.

[33] Jim Woodcock, Peter Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM computing surveys*, 41(4):19, 2009.

A Bisimilarity (Part II)

Discrete actions

Action st**:** From inference rules in table 10, to take st from γ, we must have:
$\gamma = (l, v) \models (v(xt) = Per)$ (1.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (1.a) and **Definition 21** (Rule (2)) we have:
$\phi(st) = I_{st} - Per$ and thus $\phi(st) = [0, 0]$ (1.b)
Now from inference rules in table 2, to take st we must have
$\psi = (s, \phi) \models (0 \in \phi(st))$ (1.c)
From (1.b) and (1.c) it follows that action st is possible at ψ.
We take now the action st from γ to reach the state γ'. From table 10 we have:
$\gamma' = (l', v') \models (l'(sig) = true \wedge v'(xt) = 0)$ and l' agrees with l otherwise (1.d)
We take the action st from ψ to reach the state ψ'. From table 2 we have:
$\psi' = (s', \phi') \models (s'(sig) = true \wedge \phi(st) = [Per, Per])$ and s' agrees with s otherwise (1.e)
From (1.d) and (1.e) it follows that rules (1) to (4) in **Definition 21** are satisfied by γ' and ψ', and from Sect. 6.2.2 (absence of effects on time constraints in activities) we conclude that the rule (5) in **Definition 21** is satisfied as well.
It follows that $\psi' \mathcal{R} \gamma'$.

Action sm**:** From inference rules in table 11, to take sm we must have
$\gamma = (l, v) \models (l(sig) = true \wedge l(\pi_M) = wait)$ (2.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (2.a) and **Definition 21** (Rules (1),(3)) we have at ψ:
$s(sig) = true \wedge s(\pi_M) = wait$ (2.b)
Now from inference rules in table 3, to take sm from ψ, we must have:
$\psi = (s, \phi) \models (s(sig) = true \wedge s(\pi_M) = wait)$ (2.c)
From (2.b) and (2.c) it follows that action sm is possible at ψ.
We take now the action sm from γ to reach the state γ'. From table 11 we have:
$\gamma' = (l', v') \models (l'(sig) = false \wedge l'(N) = N' \wedge l'(R) = R' \wedge l'(\pi_M) = manage)$ and l' agrees with l otherwise (2.d)
We take the action sm from ψ to reach the state ψ'. From table 3 we have:
$\psi' = (s', \phi') \models (s'(sig) = false \wedge s'(\pi_M) = manage \wedge s'(N) = N' \wedge s'(R) = R')$ and s' agrees with s otherwise (2.e)
From (2.d) and (2.e) and absence of external actions effect on the timer (Sect. 6.2.3), it follows that rules (1) to (4) in **Definition 21** are satisfied by γ' and ψ'. Also, from (2.a) and (2.d) (respect. (2.b) and (2.e)) we have $\Pi = M$ at both γ' and ψ' and thus both satisfy rule (5) in **Definition 21** ($\nexists A \in \mathcal{A} \mid s'(\Pi) = l'(\Pi) = ID_A$).
It follows that $\psi' \mathcal{R} \gamma'$.

Action lm**:** From inference rules in table 12, to take lm we must have:
$\gamma = (l, v) \models (l(\Pi) = M \wedge (l(N) \cup l(R)) \neq \varnothing)$ (3.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (3.a) and **Definition 21** (Rule (1)) we have at ψ:
$s(\Pi) = M \wedge (s(N) \cup s(R)) \neq \varnothing$ (3.b)
Now from inference rules in table 4, to take lm from ψ, we must have:
$\psi = (s, \phi) \models (s(\Pi) = M \wedge (s(N) \cup s(R)) \neq \varnothing)$ (3.c)

From (3.b) and (3.c) it follows that action lm is possible at ψ.
We take now the action lm (rule $lm.1$) from γ to reach the state γ'. From table 12 we have:
$\gamma' = (l', v') \models (l'(\Pi) = ID_{A \in \mathcal{A}} \wedge l'(\pi_A) = ether \wedge ID_A \in l'(R))$ and l' agrees with l otherwise (3.d)
We take the action lm (rule $lm.1$) from ψ to reach the state ψ'. From table 4 we have:
$\psi' = (s', \phi') \models (s'(\Pi) = ID_{A \in \mathcal{A}} \wedge s'(\pi_A) = ether \wedge ID_A \in s'(R))$ and s' agrees with s otherwise (3.e)
From (3.d) and (3.e) and absence of external actions effect on the timer (Sect. 6.2.3) it follows that rules (1) to (4) as well as rule (5) (through the clause 5.1) are satisfied by γ' and ψ', that is $\psi' \mathcal{R} \gamma'$ after taking lm (rule $lm.1$).

We take now the action lm (rule $lm.2$) from γ to reach the state γ'. From table 12 we have:
$\gamma' = (l', v') \models (l'(\Pi) = ID_{A \in \mathcal{A}} \wedge l'(\pi_A) = c_{pause} \wedge ID_A \in l'(R))$ and l' agrees with l otherwise (3.f)
We take the action lm (rule $lm.2$) from ψ to reach the state ψ'. From table 4 we have:
$\psi' = (s', \phi') \models (s'(\Pi) = ID_{A \in \mathcal{A}} \wedge s'(\pi_A) = c \neq ether \wedge ID_A \in s'(R))$ and s' agrees with s otherwise (3.g)
From (3.f) and (3.g) and absence of external actions effect on the timer (Sect. 6.2.3) it follows that rules (1) to (4) as well as rule (5) (through the clause 5.3) are satisfied by γ' and ψ', that is $\psi' \mathcal{R} \gamma'$ after taking lm (rule $lm.2$).

We take now the action lm (rule $lm.3$) from γ to reach the state γ'. From table 12 we have:
$\gamma' = (l', v') \models (l'(\Pi) = ID_{A \in \mathcal{A}} \wedge l'(\pi_A) = ether \wedge ID_A \notin l'(R))$ and l' agrees with l otherwise (3.h)
We take the action lm (rule $lm.3$) from ψ to reach the state ψ'. From table 4 we have:
$\psi' = (s', \phi') \models (s'(\Pi) = ID_{A \in \mathcal{A}} \wedge s'(\pi_A) = ether \wedge ID_A \notin s'(R))$ and s' agrees with s otherwise (3.i)
From (3.h) and (3.i) and absence of external actions effect on the timer (Sect. 6.2.3) it follows that rules (1) to (4) as well as rule (5) (through the clause 5.1) are satisfied by γ' and ψ', that is $\psi' \mathcal{R} \gamma'$ after taking lm (rule $lm.3$).

We take now the action lm (rule $lm.4$) from γ to reach the state γ'. From table 12 we have:
$\gamma' = (l', v') \models (l'(\Pi) = ID_{A \in \mathcal{A}} \wedge l'(\pi_A) = c_{pause} \wedge ID_A \notin l'(R))$ and l' agrees with l otherwise. (3.j)
We take now the action lm (rule $lm.4$) from ψ to reach the state ψ'. From table 4 we have:
$\psi' = (s', \phi') \models (s'(\Pi) = ID_{A \in \mathcal{A}} \wedge s'(\pi_A) = c \neq ether \wedge ID_A \notin s'(R) \wedge$
$(\phi'(ea) = I_{ea} \vee \phi'(fa) = I_{fa}))$ and s' agrees with s otherwise (3.k)
From (3.j) and (3.k) and absence of external actions effect on the timer (Sect. 6.2.3) it follows that rules (1) to (4) as well as rule (5) (through the clause 5.3 with $\phi'(ea) = I_{ea} \vee \phi'(fa) = I_{fa}$) are satisfied by γ' and ψ', that is $\psi' \mathcal{R} \gamma'$ after taking lm (rule $lm.4$).
Note here that taking the action τ from γ' would give the state $\gamma'' = (l'', v'')$ such that

$l''(\pi_A) = c$, $v''(x_A) = 0$ and l'' agrees with l' otherwise. We may easily deduce then that $\psi' \mathcal{R} \gamma''$ (Rule (5) through clause 5.2 with $\theta = 0$).

Action fm**:** From inference rules in table 13, to take fm we must have
$\gamma = (l, v) \models (l(\Pi) = M \wedge (l(N) \cup l(R)) = \varnothing \wedge l(\pi_M) = manage)$ (4.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (4.a) and **Definition 21** (Rules (1),(3)) we have at ψ:
$s(\Pi) = M \wedge (s(N) \cup s(R)) = \varnothing \wedge s(\pi_M) = manage$ (4.b)
Now from inference rules in table 5, to take fm from ψ, we must have:
$\psi = (s, \phi) \models (s(\Pi) = M \wedge (s(N) \cup s(R)) = \varnothing \wedge s(\pi_M) = manage)$ (4.c)
From (4.b) and (4.c) it follows that action fm is possible at ψ.
We take now the action fm from γ to reach the state γ'. From table 13 we have:
$\gamma' = (l', v') \models (l'(\pi_M) = wait)$ and l' agrees with l otherwise (4.d)
We take the action fm from ψ to reach the state ψ'. From table 5 we have:
$\psi' = (s', \phi') \models (s'(\pi_M) = wait)$ and s' agrees with s otherwise (4.e)
From (4.d) and (4.e) and absence of external actions effect on the timer (Sect. 6.2.3) it follows that rules (1) to (4) in **Definition 21** are satisfied by γ' and ψ'. Also, from (4.a) and (4.d) (respect. (4.b) and (4.e)) we have $\Pi = M$ at γ' and ψ' and thus both satisfy rule (5) in **Definition 21** ($\nexists A \in \mathcal{A} \mid s'(\Pi) = l'(\Pi) = ID_A$).
It follows that $\psi' \mathcal{R} \gamma'$.

Action ia**:** From inference rules in table 14 to take ia (rule $ia.1$ or $ia.3$) we must have, besides the existence of an outgoing ia edge from $ether$ (to $stop$ (rule $ia.1$) or to $ether$ ($ia.3$)):
$\gamma = (l, v) \models (l(\Pi) = ID_{A \in \mathcal{A}} \wedge l(\pi_A) = ether \wedge ID_A \in l(R)$ (5.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (5.a) and **Definition 21** (Rules (1), (5.1)) we have at ψ:
$s(\Pi) = ID_{A \in \mathcal{A}} \wedge s(\pi_A) = ether \wedge ID_A \in s(R)$ (5.b)
Now from inference rules in table 6, to take ia (rule $ia.1$ or $ia.3$) from ψ, we must have, besides the existence of an outgoing ia edge from $ether$ (to $stop$ (rule $ia.1$) or to $ether$ ($ia.3$)):
$\psi = (s, \phi) \models (s(\Pi) = ID_{A \in \mathcal{A}} \wedge s(\pi_A) = ether \wedge ID_A \in s(R))$ (5.c)
From (5.a), (5.b) and (5.c) and edges equivalence (Sect. 6.2.4) it follows that action ia (rule $ia.1$ or $ia.3$) is possible at ψ.
We take now the action ia (rule $ia.1$) from γ to reach the state γ'. From table 14 we have:
$\gamma' = (l', v') \models (l'(\pi_A) = stop \wedge v'(x_A) = 0 \wedge ID_A \notin l'(R))$ and l' agrees with l otherwise (5.d)
We take the action ia (rule $ia.1$) from ψ to reach the state ψ'. From table 6 we have:
$\psi' = (s', \phi') \models (s'(\pi_A) = stop \wedge ID_A \notin s'(R) \wedge (\phi' ea = I_{ea} \vee \phi'(fa) = I_{fa}))$ and s' agrees with s otherwise (5.e)
From (5.d) and (5.e) and absence of external actions effect on the timer (Sect. 6.2.3) it follows that rules (1) to (4) as well as rule (5) (through the clause 5.2 with $\theta = 0$) are satisfied by γ' and ψ', that is $\psi' \mathcal{R} \gamma'$ after taking ia (rule $ia.1$).

We take now the action ia (rule $ia.3$) from γ to reach the state γ'. From table 14 we have:
$\gamma' = (l', v') \models (l'(\pi_A) = ether \wedge l'(\Pi) = M \wedge \neg(ID_A \in l'(N) \vee ID_A \in l'(R)))$ and

l' agrees with l otherwise (5.f)
We take the action ia (rule $ia.3$) from ψ to reach the state ψ'. From table 6 we have:
$\psi' = (s', \phi') \models (s'(\pi_A) = ether \wedge s'(\Pi) = M \wedge \neg(ID_A \in s'(N) \vee ID_A \in s'(R)))$ and s' agrees with s otherwise (5.g)
From (5.f) and (5.g) and absence of external actions effect on the timer (Sect. 6.2.3) it follows that rules (1) to (5) are satisfied by γ' and ψ' ((5) is satisfied because $\Pi = M$ at both ψ' and γ' and thus $\nexists A \in \mathcal{A} \mid s'(\Pi) = l'(\Pi) = ID_A$). It follows that $\psi' \mathcal{R} \gamma'$ after taking $ia.1$ or $ia.3$.

From inference rules in table 14, to take ia (rule $ia.2$ or $ia.4$) we must have, besides the existence of an outgoing ia edge from c_{pause} (to $stop$ (rule $ia.2$) or to $ether$ ($ia.4$)):
$\gamma = (l, v) \models (l(\Pi) = ID_{A \in \mathcal{A}} \wedge l(\pi_A) = c_{pause} \wedge ID_A \in l(R)$ (5.h)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (5.h) and **Definition 21** (Rules (1),(5.3)) we have at ψ:
$s(\Pi) = ID_{A \in \mathcal{A}} \wedge s(\pi_A) = c \wedge ID_A \in s(R)$ (5.i)
Now from inference rules in table 6, to take ia (rule $ia.2$ or $ia.4$) from ψ, we must have, besides the existence of an outgoing ia edge from c (to $stop$ (rule $ia.2$) or to $ether$ ($ia.4$)):
$\psi = (s, \phi) \models (s(\Pi) = ID_{A \in \mathcal{A}} \wedge s(\pi_A) = c \neq ether \wedge ID_A \in s(R))$ (5.j)
From (5.h), (5.i), (5.j) and edges equivalence (Sect. 6.2.4) it follows that action ia (rule $ia.2$ or $ia.4$) is possible at ψ.
Now, proving that $\psi' \mathcal{R} \gamma'$ after applying rule $ia.2$ (respect. $ia.4$) is identical to proving $\psi' \mathcal{R} \gamma'$ after applying rule $ia.1$ (respect. $ia.3$).

Action fa: From inference rules in table 15, to take fa we must have, besides the existence of an outgoing fa edge from c (to c'_{pause} (rule $fa.1$) or to $ether$ ($fa.2$)):
$\gamma = (l, v) \models (l(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin l(R) \wedge l(\pi_A) = c \wedge v(x_A) > 0)$ (6.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (6.a) and **Definition 21** (Rules (1),(5.2) we have at ψ:
$s(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin s(R) \wedge s(\pi_A) = c \wedge (\phi(fa) = I_{fa} - \theta \mid \theta > 0)$ (6.b)
Now from inference rules in table 7, to take fa from ψ, we must have, besides the existence of an outgoing fa edge from c (to $c' \neq ether$ (rule $fa.1$) or to $ether$ ($fa.2$)):
$\psi = (s, \phi) \models (s(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin s(R) \wedge s(\pi_A) = c \wedge (\phi(fa) = I_{fa} - \theta \mid \theta > 0))$ (6.c)
From (6.a), (6.b), (6.c) and edges equivalence (Sect. 6.2.4) it follows that action fa is possible at ψ.
We take now the action fa (rule $fa.1$) from γ to reach the state γ'. From table 15 we have:
$\gamma = (l', v') \models (l'(\pi_A) = c'_{pause}, l'(\Pi) = M \wedge \neg(ID_A \in l'(N) \vee ID_A \in l'(R)))$ and l' agrees with l otherwise (6.d)
We take the action fa (rule $fa.1$) from ψ to reach the state ψ'. From table 7 we have:
$\psi' = (s', \phi') \models (s'(\pi_A) = c' \neq ether \wedge s'(\Pi) = M \wedge \neg(ID_A \in s'(N) \vee ID_A \in s'(R)))$ and s' agrees with s otherwise (6.e)
From (6.d) and (6.e) and absence of external actions effect on the timer (Sect. 6.2.3) it follows that rules (1) to (4) in **Definition 21** are satisfied by γ' and ψ'. Also, we have $\Pi = M$ at γ' and ψ' and thus both satisfy rule (5) in **Definition 21** ($\nexists A \in \mathcal{A} \mid s'(\Pi) = l'(\Pi) = ID_A$).

It follows that $\psi' \mathcal{R} \gamma'$.

We take now the action fa (rule $fa.2$) from γ to reach the state γ'. From table 15 we have:
$(l', v') \models (l'(\pi_A) = ether \wedge l'(\Pi) = M \wedge \neg(ID_A \in l'(N) \vee ID_A \in l'(R)))$ and l' agrees with l otherwise (6.f)
We take now the action fa (rule $fa.2$) from ψ to reach the state ψ'. From table 7 we have:
$\psi' = (s', \phi') \models (s'(\pi_A) = ether \wedge s'(\Pi) = M \wedge \neg(ID_A \in s'(N) \vee ID_A \in s'(R)))$ and s' agrees with s otherwise (6.g)
From (6.f) and (6.g) and absence of external actions effect on the timer (Sect. 6.2.3) it follows that rules (1) to (4) in **Definition 21** are satisfied by γ' and ψ'. Also, we have $\Pi = M$ at γ' and ψ' and thus both satisfy rule (5) in **Definition 21** ($\nexists A \in \mathcal{A} \mid s'(\Pi) = l'(\Pi) = ID_A$).
It follows that $\psi' \mathcal{R} \gamma'$.

Action ea**:** From inference rules in table 16, to take ea (rule $ea.1$) we must have:
$\gamma = (l, v) \models (l(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin l(R) \wedge l(\pi_A) = ether)$ (7.a)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (7.a) and **Definition 21** (Rules (1),(5.1)) we have at γ:
$s(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin s(R) \wedge s(\pi_A) = ether$ (7.b)
Now, from inference rules in table 8, to take ea (rule $ea.1$) from ψ, we must have:
$\psi = (s, \phi) \models (s(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin s(R) \wedge s(\pi_A) = ether)$ (7.c)
From (7.b), (7.c) it follows that action ea is possible at ψ.
We take now the action ea (rule $ea.1$) from γ to reach the state γ'. From table 16 we have:
$\gamma' = (l', v') \models (l'(\pi_A) = start \wedge v'(x_A) = 0)$ and l' agrees with l otherwise (7.d)
We take now the action ea (rule $ea.1$) from ψ to reach the state ψ'. From table 8 we have:
$\psi' = (s', \phi') \models (s'(\pi_A) = start \wedge (\phi'(ea) = I_{ea} \vee \phi'(fa) = I_{fa}))$ and s' agrees with s otherwise (7.e)
From (7.d) and (7.e) and absence of external actions effect on the timer (Sect. 6.2.3) it follows that rules (1) to (4) as well as rule (5) (through the clause 5.2 with $\theta = 0$) are satisfied by γ' and ψ'.
It follows that $\psi' \mathcal{R} \gamma'$.

From inference rules in table 16, to take ea (rule $ea.2$) we must have, besides the existence of an outgoing ea edge from c:
$\gamma = (l, v) \models (l(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin l(R) \wedge l(\pi_A) = c \neq ether \wedge v(x_A) > 0)$ (7.f)
Additionally, we know that $\psi \mathcal{R} \gamma$, then from (7.f) and **Definition 21** (Rules (1),(5.2)) we have at ψ:
$s(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin s(R) \wedge s(\pi_A) = c \neq ether \wedge$
$\wedge(\phi(ea) = I_{ea} - \theta \vee \phi(fa) = I_{fa} - \theta)$ with $\theta = v(x_A)$ (7.g)
Now from inference rules in table 8, to take ea (rule $ea.2$) from ψ, we must have, besides the existence of an outgoing ea edge from c:
$\psi = (s, \phi) \models (s(\Pi) = ID_{A \in \mathcal{A}} \wedge ID_A \notin s(R) \wedge s(\pi_A) = c \neq ether \wedge \phi(ea) = I_{ea} - \theta \mid \theta > 0)$ (7.h)
From (7.f), (7.g) and (7.h) and edges equivalence (Sect. 6.2.4) it follows that action ea

(rule $ea.2$) is possible at ψ.
We apply now the rule $ea.2$ from γ to reach the state γ' (table 16) then from ψ to reach the state ψ' (table 8). The proof that $\psi'\mathcal{R}\gamma'$ is similar to that when taking $ea.1$ (with replacing $start$ by c'.

Time actions From inference rules in table 18, to take d (rule $d.1$) we must have:
$(l, v) \models (v(xt) < Per \wedge l(sig) = false \wedge l(\pi_M) = wait)$ (8.a)
Additionally, we know that $\psi\mathcal{R}\gamma$, then from (8.a) and **Definition 21** (Rules (1),(2),(3)) we have at ψ:
$\phi(st) = I_{st} - a \wedge s(sig) = false \wedge s(\pi_M) = wait$ s.t. $a = v(xt)$ (8.b)
Now from inference rules in table 9, to take d (rule $d.1$) from ψ, we must have:
$\psi = (s, \phi) \models ((\phi(st) = I_{st} - a \mid a < Per) \wedge s(sig) = false \wedge s(\pi_M) = wait)$ (8.c)
From (8.a), (8.b) and (8.c) it follows that action d ($d \in]0, Per - a]$) is possible at ψ.
We take now the action d ($d \in]0, Per - a]$) from γ to reach the state γ'. From table 18 (rule $d.1$) we have:
$\gamma' = (l, v') \models (\forall x \in X : v'(x) = v(x) + d)$, which means $v'(xt) = a + d$ (8.d)
We take now the action d from ψ to reach the state ψ'. From table 9 (rule $d.1$) we have:
$\psi' = (s, \phi') \models (\phi'(st) = I_{st} - a - d)$ (8.e)
From (8.d) and (8.e) it follows that rules (1), (2 with $\theta = a + d$), (3), (4), and (5) (the manager has the control and thus $\nexists A \in \mathcal{A} \mid s(A) = l(A) = ID_A$) are satisfied by γ' and ψ'.
It follows that $\psi'\mathcal{R}\gamma'$.

From inference rules in table 18 (rule $d.2$), to take d we must have:
$(l, v) \models (v(xt) < Per \wedge l(\Pi) = ID_A \wedge$
$l(\pi_A) = c \neq ether \wedge ID_A \notin R \wedge v(x_A) < W(c))$ (8.f)
Additionally, we know that $\psi\mathcal{R}\gamma$, then from (8.f) and **Definition 21** (Rules (1),(2),(5.2)) we have at ψ:
$\phi(st) = I_{st} - a \mid a < Per \wedge s(\Pi) = ID_A \wedge$
$s(\pi_A) = c \neq ether \wedge ID_A \notin s(R) \wedge (\phi(ea) = I_{ea} - b \vee \phi(fa) = I_{fa} - b \mid b < W(c)))$
s.t. $v(x_A) = b$ and $v(xt) = a$ (8.g)
Now from inference rules in table 9, to take d (rule $d.2$) from ψ, we must have:
$\psi = (s, \phi) \models (\phi(st) = I_{st} - a \mid a < Per \wedge s(\Pi) = ID_A \wedge$
$s(\pi_A) = c \neq ether \wedge ID_A \notin s(R) \wedge (\phi(ea) = I_{ea} - b \vee \phi(fa) = I_{fa} - b \mid b < W(c)))$
(8.h)
From (8.f), (8.g) and (8.h) it follows that action d ($d \in]0, min(W(c) - b, Per - a)]$) is possible at ψ.
We take now the action d ($d \in]0, min(W(c) - b, Per - a)]$) from γ to reach the state γ'. From table 18 (rule $d.2$) we have:
$\gamma' = (l, v') \models (\forall x \in X : v'(x) = v(x) + d)$, which means $(v'(xt) = a + d \wedge v'(x_A) = b + d)$ (8.i)
We take the action d from ψ to reach the state ψ'. From table 9 (rule $d.2$) we have:
$\psi' = (s, \phi') \models (\phi'(st) = I_{st} - a - d \wedge (\phi'(ea) = I_{ea} - b - d \vee \phi'(fa) = I_{fa} - b - d))$ (8.j).
From (8.i) and (8.j) it follows that rules (1), (2 with $\theta = a + d$), (3), (4), and (5) (through the clause 5.2 with $\theta = b + d$) are satisfied by γ' and ψ'.

It follows that $\psi' \mathcal{R} \gamma'$.

We have thus proven that for all discrete and time actions, if $\psi \mathcal{R} \gamma$ and action act is possible from γ s.t. $\gamma \xrightarrow{act} \gamma'$, then the same action act is possible from ψ s.t. $\psi \xrightarrow{act} \psi'$ and $\psi' \mathcal{R} \gamma'$. It follows that Ψ **(weakly) time simulates** Γ.