Investigating Process Algebra Models to Represent Structured Requirements for Time-sensitive CPS

Mathilde Arnaud [★]   Boutheina Bannour [★]   Arnault Lapitre [★]   Guillaume Giraud [☆]

[★] Université Paris-Saclay, CEA, List   [☆] PES R&D Department, RTE

Abstract

Cyber-Physical Systems (CPS) contain complex computational components that control physical entities. The design of these components must take into account the real-time and concurrent nature of these systems. Formulating requirements that describe CPS behaviors precisely, ruling out misunderstandings, is a crucial yet difficult endeavor. To increase trust in the requirements, formal methods can be used to check relevant properties of the requirements. We investigate a process algebra to capture real-time behaviors and concurrency in CPS requirements in order to automate their analysis. We use a structured natural language to first express CPS requirements: this takes into account current practice, indeed requirements should be easily writable as well as graspable by stakeholders with various points of view and ease communication among them. At the same time, requirements analysis using simulation or formal validation is possible by taking advantage of the requirements structure. We discuss translation from the structured requirements into the process algebra to automate the overall process. Our approach is implemented and is illustrated by an example issued from CPS4EU project [1].

1 Introduction

Early validation of Cyber-Physical Systems (CPS) requires the consolidation of requirements, a tedious task due to the nature of CPS behavior. Indeed, physical devices have to be reactive, available and resilient within acceptable time frames, and their control logic can be quite complex. Cross-checking CPS industrial requirements, still mostly expressed in natural language, presents a major challenge, as missing or contradictory requirements can create a costly misunderstanding in the CPS development process. Formal methods can help meet this challenge by validating CPS requirements. In order to apply automated formal analyses, requirements need to be specified in a precise and formal way, through the use of patterns for instance. Fill-in templates facilitate clearer specification of event-driven, state-driven system behaviors. The approach described in this paper specifies CPS requirements following EARS [13] templates, in accordance with recommendations from the International Council on Systems Engineering. Real-time details are introduced to refine event-driven, state-driven system behaviors. An transformation of such requirements into a Process Algebra (PA) is proposed. This process algebra has been implemented in the model-based symbolic execution tool DIVERSITY[10]. Comparatively to other PA available tools[2], we support real-time behavior modeling. Likewise, we go a step further by considering real-time aspects compared to related works on formalizing requirement using PA until recently [6, 1, 14]. The implementation of PA benefits from existing sat-based techniques integrated in DIVERSITY to support models of timed symbolic automata [3]. Via the transformation, behaviors of CPS systems specified by the requirements can be explored in the tool. Edition and transformation of the requirements are prototyped as a web application using Jupyter Notebook environment. A simplified version of a CPS4EU case study illustrates the proposed approach throughout the paper.

2 Requirement specification

Illustrative example. We illustrate our approach by showing how it can be applied on a real-world use case coming from CPS4EU : electrical networks involving intermittent energy sources. To avoid overload without raising the overall network capacity, it is necessary to manage dynamically the flow of electricity through levers such as batteries or production modulation. Which mechanisms to trigger must be determined very quickly and this role must therefore be entrusted to a software component called NAZA.

We used our approach to analyze NAZA functional requirements. For the purpose of illustration, we choose a subset of requirements : R1 to R6A in Tab.1, written in Structured

---

[2]Some references and tooled process algebra are CADP/LOTOS (https://cadp.inria.fr/) and FDR/CSP (https://cocotec.io/fdr/).

$R1$	**every** 5 seconds, **the** NAZA Core **shall** calculate levers setpoints
$R2$	**when** new levers setpoints have been determined **upon** levers setpoints calculation ($R1$), **the** NAZA Core **shall** determine common levers by using consensus
$R3$	**every** 5 seconds **when** consensus **upon** common levers determination ($R2$), **the** NAZA Core **shall** send batteries setpoints
$R4$	**every** 5 seconds **when** consensus **upon** common levers determination ($R2$), **the** NAZA Core **shall** send topological orders
$R5$	**every** 5 seconds **when** consensus **upon** common levers determination ($R2$), **the** NAZA Core **shall** send modulation orders
$R6A$	**if** no result **upon** levers setpoints calculation ($R1$), **the** NAZA Core **shall** execute backup algorithm **within** [10,60] seconds
$R6B$	**if** no result **upon** levers setpoints calculation ($R1$), **while** in nominal mode, **then the** NAZA Supervisor **shall enter** in backup mode
$R7$	**when entering** in $backup$ mode, **the** NAZA Supervisor **shall** execute backup algorithm **within** [10,60], **and return** in nominal mode
$R8$	**when** new setpoints **upon** levers setpoints calculation ($R1$), **while** in backup mode, **the** NAZA Supervisor **shall enter** in nominal mode.

Table 1: Excerpt requirements on levers setpoints calculation in NAZA (Nouveaux Automates de Zones Adaptatifs)

Natural Language as presented below. The NAZA automaton is in charge of computing levers setpoints (cf R1). When the computation is successful, it then uses consensus (cf R2) and sends the results to middleware (cf R3, R4, R5). When the computation fails, it must launch a backup logigram algorithm (cf R6A). Our analysis revealed possible deadlocks and we proposed to replace requirement R6A by requirements R6B, R7 and R8, using two modes ($nominal$ and $backup$) to express the behavior more precisely.

Structured Natural Language. We have structured requirement statements by using a grammar based on EARS [13]. A user-defined glossary, tailored to the needs of the requirement engineer, defines systems, triggers, and also equivalence for ease of use (e.g. "calculate levers setpoints"/"levers setpoints calculations"). To prevent ambiguity arising from the use of synonyms, we favor the use of repetitions of expressions in the glossary (e.g. R3, R4, R5). This makes requirements as simple as possible and thus preserves readability and unity. Each requirement statement is expressed by a -possibly complex- precondition, followed by a realization, which specifies the action of the system.

Preconditions. *Nominal* and *unwanted* behavior requirements are initiated when a triggering event occurs. They are built respectively with keywords **when** (e.g. R7) and **if** (e.g. R6B). *State-driven* requirements are active while the system is in a defined state and are built with keyword **while** (e.g. R6B). We introduce details to enhance the sequencing of system behaviors: they can be triggered periodically, subsequently to other behaviors, or within some time slot. *Periodic behaviors* are expressed through pattern "**every** ⟨ period ⟩" (e.g. R1, R3, R4, R5). *Context execution* can be detailed through two constructs : "**within** ⟨ timing interval ⟩", and "**upon** ⟨ system response ⟩", specifying that the behavior happens subsequently to some other behavior. R6B is an example of a combined use of these constructs, demonstrating that requirements can be complex and use several of these constructs at the same time.

3   Target process algebra

Time datatype, actions and modes. Clocks are typed in a dense time domain T isomorphic to the set of positive rational numbers $\mathbb{Q}_+$. Given a set of clocks Clk, a clock valuation v is a mapping $v : Clk \to T$. The set $\mathcal{F}(Clk)$ of clock formulas is built up recursively out of logical conjunction and atomic formulas of the form $True$, $False$, $clk \bowtie d$, where d is a constant duration (typed in T) and $\bowtie \in \{<, \leq, >, \geq\}$. The set of clock invariants $\mathcal{I}(Clk)$ is defined by conjunctions of formulas of the form $clk \bowtie d$, where $\bowtie \in \{<, \leq\}$. Let Act be a set of actions which contains the silent action $\tau \in Act$, and $A \subseteq Act \setminus \{\tau\}$ be a partition $I \cup O$. Elements a of I (resp. of O) are called inputs and denoted by $?a$ (resp. called outputs and denoted by $!a$). In a parallel composition, inputs and outputs can synchronize resulting in τ. We denote $\overline{?a} = !a$ (and vice versa). Let M be a set of modes with initial mode $m_0 \in M$.

Processes. A process is defined by the following syntax:
$$P \quad ::= \quad \sum_{i \in I} \alpha_i.P_i \mid inv(\psi).P \mid nil \mid P_1|P_2 \mid K$$
$\alpha ::= (m, \phi, a, R, m')$ are the basic building blocks of the syntax. They are described by an enabling mode $m \in M$, an enabling clock formula $\phi \in \mathcal{F}(Clk)$, an action $a \in Act$, a set of clocks $R \subseteq Clk$ to be reset, and a target mode $m' \in M$ to evolve into. Some of these elements can be dropped. For instance (ϕ, a, R) denotes that enabling mode can match any arbitrary mode and that no mode change is to be made. The construct $inv(\psi).P$ defines a clock invariant $\psi \in \mathcal{I}(Clk)$ that has to be satisfied on time passing for the execution of the process P. This notion is borrowed from timed (and hybrid) automata and requires some technical handling at the evaluation of the process. All other constructs are classic [5]: the empty process nil, action prefixing $\alpha_i.P_i$, non-deterministic choice $\sum_{i \in I} \alpha_i.P_i$, parallel composition $P_1|P_2$ with possible synchronization of input / output actions, the process constant $K \overset{def}{=} P$ for recursive definition.

Process execution. The key idea is to ensure that time

progress cannot invalidate either of the local invariants of parallel processes. We introduce a construct of global invariants at evaluation that will be updated upon the evaluation of the left or right processes in a parallel composition. A global invariant or g-invariant is defined by the syntax: $\Psi ::= \psi \mid \Psi \wedge_{\_} \Psi \mid \Psi \wedge_L \Psi \mid \Psi \wedge_R \Psi$, with $\psi \in \mathcal{I}(Clk)$. We define functions L, R and f on g-invariants that return respectively the left side of the g-invariant, the right side, and the formula denoting conditions at the time of evaluation. If Ψ is in a decomposed form $\Psi_1 \wedge_X \Psi_2$ with $X \in \{\_, L, R\}$ then $L(\Psi)$, $R(\Psi)$ and $f(\Psi)$ denote Ψ_1, Ψ_2 and $f(\Psi_1) \wedge f(\Psi_2)$ respectively, otherwise $L(\psi)$, $R(\psi)$ and $f(\psi)$ are ψ. The process execution is defined up to an execution context $ec = (m, v, \Psi)$ which represents the necessary information to perform an execution step, namely the current mode m, the current valuation v of clocks and the current g-invariant Ψ to be applied. Operational rules of the execution are defined as follows:

Rule ATOM

$$\vdash (m, \phi, a, R, m').P \quad (m, v, \Psi) \overset{a}{\to} P \quad (m', v', \Psi)$$

with $v_0 = v[clk \to clk + d, clk \in Clk], d \in T, v_0 \vDash \phi \wedge f(\Psi)$ and $v' = v_0[clk \to 0, clk \in R]$.

Rule INV

$$\vdash inv(\psi).P \quad (m, v, \Psi) \overset{\tau}{\to} P \quad (m, v, \Psi')$$

with $v \vDash f(\Psi)$, and $\Psi' = ginv\_upd(\Psi, \psi)$.

Rule SUM

$$\alpha_i.P_i \quad ec \overset{a}{\to} P_i' \quad ec' \vdash \sum_{i \in I} \alpha_i.P_i \quad ec \overset{a}{\to} P_i' \quad ec'$$

Rule CONST

$$P \quad ec \overset{a}{\to} P' \quad ec' \vdash K \quad ec \overset{a}{\to} P' \quad ec'$$

with $K \overset{def}{=} P$.

Rule PAR1L

$$P_1 \quad (m, v, L(\Psi) \wedge_L ginv(P_2, R(\Psi))) \overset{a}{\to} P_1' \quad (m', v', \Psi_1')$$
$$\vdash P_1 | P_2 \quad (m, v, \Psi) \overset{a}{\to} P_1' | P_2 \quad (m', v', \Psi')$$

with $\Psi' = ginv\_upd(L(\Psi) \wedge_L R(\Psi), \Psi_1')$

Rule PAR2L

$$P_1 \quad (m, v, L(\Psi) \wedge_L ginv(P_2, R(\Psi))) \overset{a}{\to} P_1' \quad (m', v_1', \Psi')$$
$$P_2 \quad (m, v, ginv(P_1, L(\Psi)) \wedge_R R(\Psi)) \overset{\bar{a}}{\to} P_2' \quad (m', v_2', \Psi')$$
$$\vdash$$
$$P_1 | P_2 \quad (m, v, \Psi) \overset{\tau}{\to} P_1' | P_2' \quad (m', v', \Psi)$$

with $v'(clk) = 0$ if $v_1'(clk) = 0 \vee v_2'(clk) = 0$, else $v'(clk) = v_1'(clk) = v_2'(clk)$, for any clock $clk \in Clk$.

The process execution is inductively defined on the form of the process term. Action prefixing $(m, \phi, a, R, m').P_i$ evolves to P_i under the constraint that time elapsing is compatible with its clock formula ϕ and formula of current g-invariant $f(\Psi)$. In case of invariant definition $inv(\psi).P$, then function $ginv\_upd$ is called to update the relevant side of the g-invariant for next action executions in P and other parallel processes if any. In fact, the case of parallel com-

position is the most subtle, other rules are classic [5]. It uses a function $ginv$ which computes the formula of a g-invariant of process P given a current g-invariant Ψ (of an ec). Concerned part of Ψ is returned in case no new invariant is encountered. For clarity's sake, functions $ginv\_upd$ and $ginv$ are defined using the Pattern-Matching notation of OCaml [3] as follows:

$ginv_upd(\Psi, \Psi') = $ **match** Ψ **with**
  $\mid \quad \psi \to \Psi' \mid \quad \Psi_1 \wedge_{\_} \Psi_2 \to \Psi' \mid \quad \Psi_1 \wedge_R \Psi_2 \to \Psi_1 \wedge_{\_} R(\Psi')$
  $\mid \quad \Psi_1 \wedge_L \Psi_2 \to L(\Psi') \wedge_{\_} \Psi_2$

$ginv(P, \Psi) = $ **match** P **with**
  $\mid \quad \sum_{i \in I} \alpha_i.P_i \quad \to \quad f(\Psi)$
  $\mid \quad nil \quad \to \quad f(\Psi)$
  $\mid \quad inv(\psi).P \quad \to \quad \psi$
  $\mid \quad P_1 | P_2 \quad \to \quad ginv(P_1, L(\Psi)) \wedge ginv(P_2, R(\Psi))$
  $\mid \quad K \quad \to \quad ginv(P)$ if $K \overset{def}{=} P$

An illustrative execution of two parallel processes is given in Fig.1. It shows how actions a and b will be interleaved in the presence of invariants. From context $ec_0 = (m_0, [clk \to 0], \Psi_0)$ with $\Psi_0 = True$, both left process P_1 and right process P_2 are evaluated. P_2 can evolve into P_2' which allows to reach context ec_3. P_1 cannot be executed, as rule $PAR1L$ requires g-invariant $L(\Psi_0) \wedge_L ginv(P_2, R(\Psi_0))$ to be satisfied for P_1 to be executed, with $L(\Psi_0) = True$ and $ginv(P_2, R(\Psi_0)) = clk < 1$. The execution of P_1 is enabled by formula $clk = 1$, which cannot be satisfied at the same time as g-invariant $clk < 1$ at this point of execution. In the evaluation, once the g-invariant $True \wedge_L clk < 1$ is applied on some process (here left), it is returned in a neutral form $True \wedge_{\_} clk < 1$. Later on, from context ec_3, the execution of process P_1 becomes possible: rule $PAR1L$ applies as g-invariant becomes $True \wedge_L True$ (previously $True \wedge_L clk < 1$) which allows time elapsing with any delay.

$$P_1 | P_2 \xrightarrow{\quad b \quad} P_1 | P_2' \xrightarrow{\quad a \quad}$$

$([clk = 1], a)$
  $\mid$
$inv(clk < 1)$
$.([1/2 \le clk < 1], b)$
$.inv(True)$
  Ψ_0 global invariant of ec_0
    $True$
    $ginv(P_2, R(\Psi_0))$
    $clk < 1$

$([clk = 1], a)$
  $\mid$
$inv(True)$

Ψ_3 global invariant of ec_3
  $True \wedge_{\_} clk < 1$
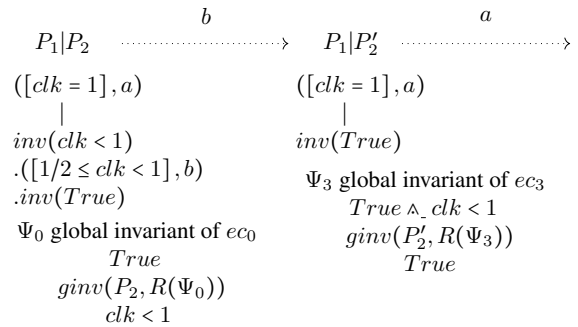 $ginv(P_2', R(\Psi_3))$
    $True$

Figure 1: Parallel execution

The transformation process described in Section 4 generates the NAZA processes given in Fig.2. We indicate each time the tuple of requirement identifiers that allowed the inference. Fig.4 is a graphical view of the NAZA Core process obtained from requirements $(R1, R2, R3, R4, R5, R6A)$.

$$Core \stackrel{def}{=}$$
$$inv(clk_1 \leq 5).calc\_setpoints$$
$$.(new\_setpoints.determine\_common\_levers. consensus$$
$$.(send\_batteries\_setpoints.([clk_1 = 5], \{clk_1\}).Core$$
$$|(send\_topological\_orders.([clk_1 = 5], \{clk_1\}).Core$$
$$|send\_modulation\_orders.([clk_1 = 5], \{clk_1\}).Core))$$
$$\boldsymbol{+(no\_result, \{clk_2\}).inv(clk_1 \leq 5 \wedge clk_2 \leq 60)}$$
$$\boldsymbol{.([10 \leq clk_2 \leq 60], execute\_backup\_algorithm)}$$
$$.inv(clk_1 \leq 5).([clk_1 = 5], \{clk_1\}).Core$$
$$\hspace{4cm}(R1, R2, R3, R4, R5, R6A)$$

$$Core' \stackrel{def}{=}$$
$$inv(clk_1 \leq 5).calc\_setpoints$$
$$.(!new\_setpoints.determine\_common\_levers. consensus$$
$$.(send\_batteries\_setpoints.([clk_1 = 5], \{clk_1\}).Core'$$
$$|(send\_topological\_orders.([clk_1 = 5], \{clk_1\}).Core'$$
$$|send\_modulation\_orders.([clk_1 = 5], \{clk_1\}).Core'))$$
$$+!no\_result.([clk_1 = 5], \{clk_1\}).Core'$$
$$\hspace{4cm}(R1, R2, R3, R4, R5, R6B)$$

$$Supervisor \stackrel{def}{=}$$
$$([nominal], ?no\_result, \{clk_2\}, \triangleright backup).Supervisor$$
$$+[backup].inv(clk_2 \leq 60)$$
$$.([10 \leq clk_2 \leq 60], execute\_backup\_algorithm, \triangleright nominal)$$
$$.inv(True).Supervisor$$
$$+([backup], ?new\_setpoints, \triangleright nominal).Supervisor$$
$$\hspace{4cm}(R6B, R7, R8)$$

Figure 2: NAZA processes

Exploration For any finite sequence of $P_0 \xrightarrow{a_0} P_1, \cdots, P_{n-1} \xrightarrow{a_n} P_n$, we note $P_0 \xrightarrow{a_1, \cdots, a_n} P_n$, or simply $P_0 \xrightarrow{*} P_n$. Our operational approach to exploration relies on small steps of the form $P \xrightarrow{a} P'$ which corresponds to the execution in a process P of an action a. This leads to a new process P' that synthesizes possible executions that may occur after action a. Potential *deadlocks* can be detected typically for a sequence $P_0 \xrightarrow{*} P_n$ such that no successor process P_{n+1} can be computed from P_n. A deadlock arises in process $Core$ (because of $R6A$). The execution of the backup algorithm in $Core$ is constrained by the formula $10 \leq clk_2 \leq 60$ which is not compatible with the formula of g-invariant $clk_1 \leq 5 \wedge clk_2 \leq 60$. The ambiguity comes from missing requirements on some parallel execution which is in charge of the backup algorithm, unconstrained by the period of 5 seconds. It is hereafter specified by a dedicated process $Supervisor$ which executes in parallel with the new $Core'$. The full NAZA case study introduces requirements for an additional complex process Acquisition, itself divided into a number of parallel processes with various periods: every 10 seconds, the NAZA Acquisition shall request new datapoints from middleware and transmit them to NAZA Core; if no data is received within 10 minutes upon datapoints request, the NAZA Supervisor shall enter in fault mode; every 1 minutes, the NAZA Acquisition shall request new network situation from upper level and transmit them to NAZA Core; etc. We have also applied exploration with the aim of identifying the so-called zeno executions [16], i.e., executions in timed systems with an unbounded number of ac-

(a) parallel responses

(b) non-deterministic triggers

(c) periodic repetition   (d) time constraint
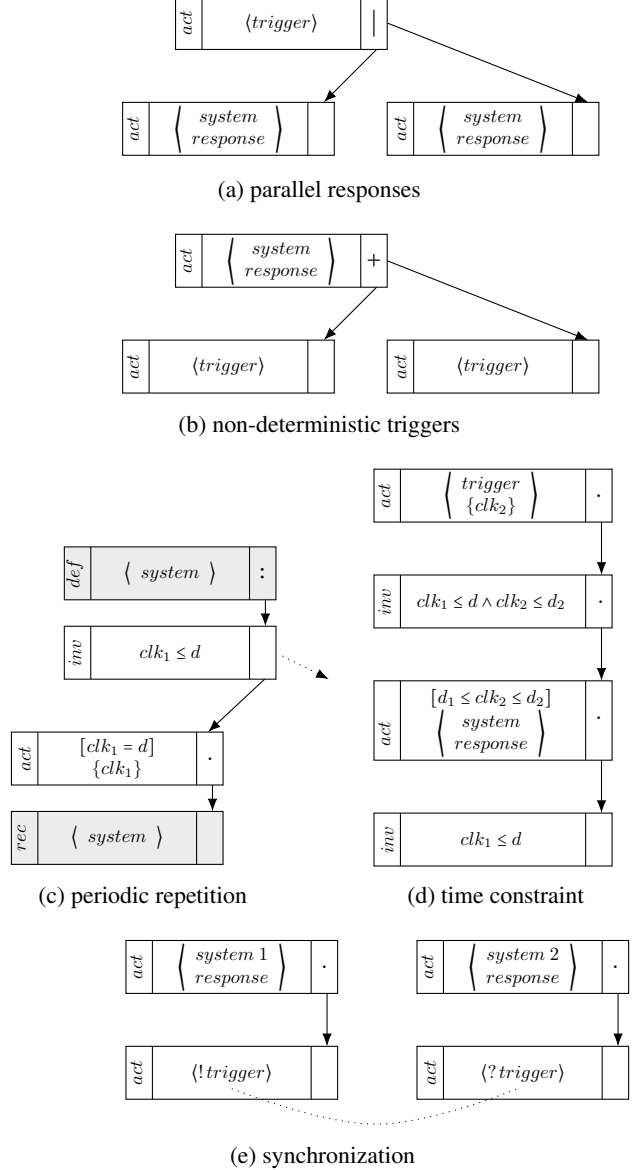
(e) synchronization

Figure 3: Transformation patterns

tions executed in a bounded length of time. The NAZA requirements should not allow such behaviors, as they are not possible in practice. By applying small-steps $P_0 \xrightarrow{a_1, \cdots, a_n} P_n$ loops ($P_n = P_0$) are checked to be non-zeno, i.e., there exists a clock clk and some $i, j \leq n$ such that clk is reset in step i and clk is bounded from below $\epsilon < clk$ in step j. Overall, the exploration of processes helped provide a good understanding of the NAZA requirements and assisted their refinement.

4 Transformation

We outline next the main transformation patterns into process algebra. System responses sharing the same trigger are composed in parallel (Fig.3a); and triggers can be non-
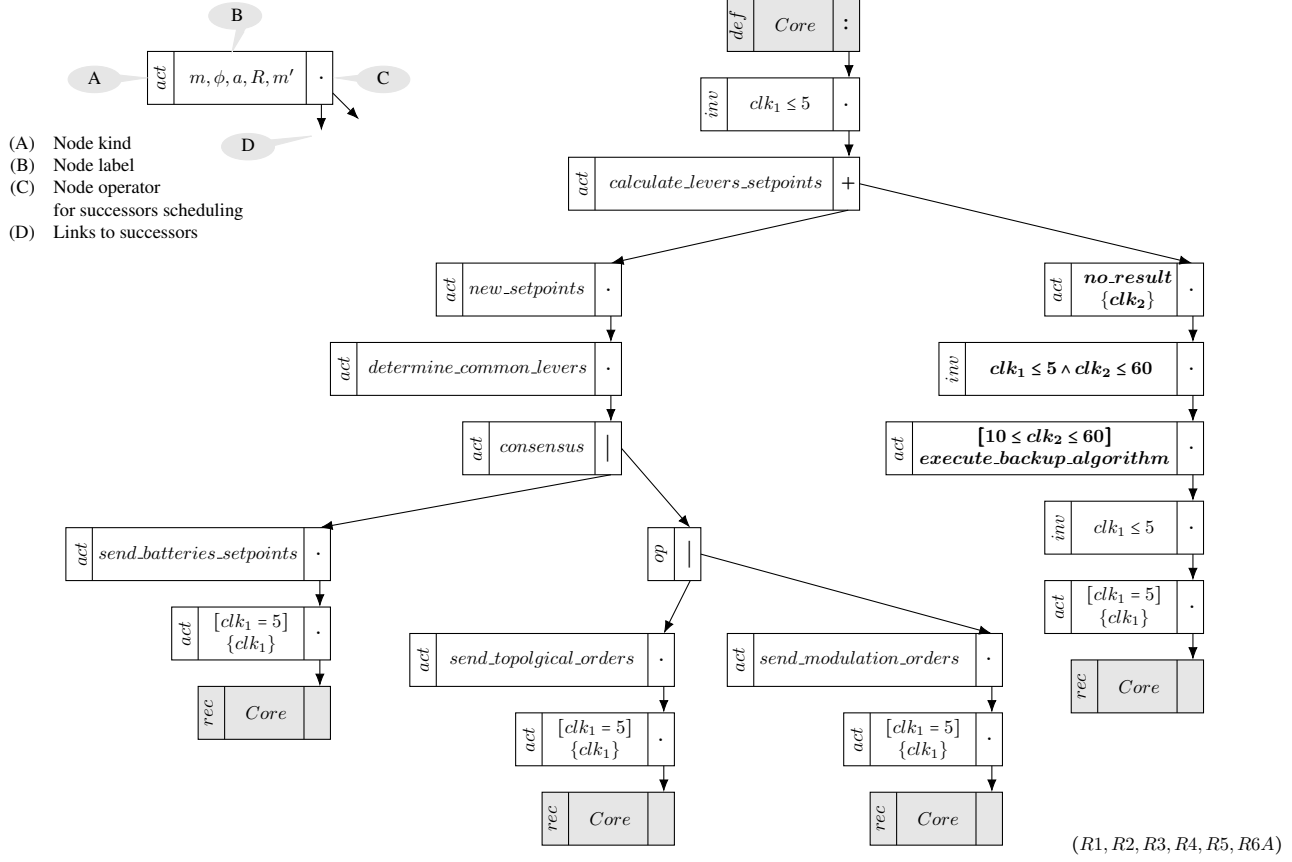
Figure 4: Graphical view of a NAZA process

determistically produced upon a system response (Fig.3b). Each (sub-)system is assumed to have an implicit initialization upon which its behavior occurs. Repetitive behavior of sub-system is also assumed with a topmost recursion (Fig.3c). It can be associated a period d : time is constrained during the iteration through an invariant of the form $inv(clk_1 \le d)$, and at the end system clock clk_1 is reset through $([clk_1 = d], \{clk_1\})$. A synchronization together with corresponding input / output actions is inferred if a sub-system response is triggered by some other sub-system behavior (Fig.3e). When a system response occurs within an interval $[d_1, d_2]$, a dedicated clock clk_2 is used to encode such constraint $(d_1 \le clk_2 \le d_2)$ associated with an invariant (re-)definition $inv(clk_1 \le d \land clk_2 \le d_2)$, which sets an upper bound d_2 on time elapsing before the response occurs (Fig.3d). State-driven triggers/responses are transformed by various patterns involving enabling modes or mode change constructs of the process algebra.

5 Related Work

Requirements are used as a tool to ensure sound communication between stakeholders for the successful design of the system [15]. Thus, specifying good requirements is important to develop qualitative products that can satisfy user's needs [4]. Cross-checking CPS industrial requirements, mostly expressed in natural language, presents a major challenge, that formal methods can help meet by validating CPS requirements.

Analyzing requirements written as natural language and getting guarantees of non-ambiguity, completeness, consistency would be ideal. Unfortunately, obtaining formal guarantees from natural language requirements is difficult, as they often lack a structure that would allow to apply formal methods. Some works explore nonetheless the possibilities for extracting useful information from natural language documents, for instance [11] automates a process based on use case documents. The authors use dependency parsing techniques to automatically generate activity diagrams describing use case flow. The input writing style for use case is fixed, so use cases may have to be rewritten according to this style, and some other steps in the method may require manual intervention. The authors provide algorithms to automatically check some structural defects they identify, but results include false positives and false negatives due to the intrinsic ambiguity of natural language. For stronger assurances, other works focus on what formal methods offer.

Several formal methods to validate requirements can be used. We focus on those that take as input natural language requirements or that address real-time, concurrent systems as our goal is to validate CPS requirements. The research problem addressed in [7] is the automatic generation of timed state-rich formal models from natural-language specifications to support test generation. Requirements are expressed according to a controlled language called SysReq-CNL, where requirements have the form of action statements guarded by conditions. The approach presented in [9] aims to check some properties or real-time requirements : rt-consistency, consistency and vacuity. Requirements must be written following a tightly constrained English grammar closely related to LTL with the underlying Duration Calculus semantics. Authors of [14] present a formalisation of requirements into a process algebra supported by the tool FDR, used to generate test cases. This work uses a strongly controlled language as input, and requirements may contain data but not time.

Filling the gap between natural language and formal methods is our goal in this paper. In this perspective, a trade-off must be found between, on the one hand, possibly difficult to master, strongly constrained requirements [17], and on the other hand, natural language requirements, that must be transformed into formal languages [2] and may lose the intended meaning in the process [8]. Fill-in templates facilitate clearer specification of event-driven, state-driven system behaviors. Automatically writing and analyzing such semi-formal requirements is still a challenge, especially when taking time into account. Typically, [12] synthesizes from EARS the logic of CPS controllers, however timing details are not formally analyzed.

6   Conclusion

This paper investigate a real-time process algebra to represent structured natural language requirements. Compositional modeling using process algebra provides powerful constructs to build larger processes from smaller ones specified by the unitary structured requirements through transformation techniques. The proposed approach is implemented, which enables to explore the resulting model and thus to better understand the real-time behaviors and concurrency implied by the requirements. These first results have to be consolidated on larger experiments. A possible continuation of this work is to develop refinement or bisimulation methods for the process algebra in order to assist requirements evolution and clarification.

References

[1] R. Almeida, S. Nogueira, and A. Sampaio. Automatic Test Case Generation for Concurrent Features from Natural Language Descriptions. In *SBMF*. Springer, 2018.

[2] J. Badger, D. Throop, and C. Claunch. VARED: Verification and analysis of requirements and early designs. In *RE*, 2014.

[3] B. Bannour, J. Escobedo, C. Gaston, and P. Le Gall. Off-line Test Case Generation for Timed Symbolic Model-Based Conformance Testing. In *ICTSS*. Springer, 2012.

[4] A. Bennaceur, T. Tun, Y. Yu, and B. Nuseibeh. Requirements engineering. In *Handbook of Software Engineering*. Springer, 2019.

[5] J. Bergstra, A. Ponse, and S. Smolka, editors. *Handbook of Process Algebra*. Elsevier, 2001.

[6] G. Cabral and A. Sampaio. Formal Specification Generation from Requirement Documents. *Elec. Notes Theor. Comput. Sci.*, 2008.

[7] G. Carvalho, A. Cavalcanti, and A. Sampaio. Modelling timed reactive systems from natural-language requirements. *Formal Aspects Comput.*, 2016.

[8] J. Greghi, E. Martins, and A. Carvalho. Semi-automatic generation of extended finite state machines from natural language standard documents. In *DSN*, 2015.

[9] V. Langenfeld, D. Dietsch, B. Westphal, J. Hoenicke, and A. Post. Scalable analysis of real-time requirements. In *Int. Conf. RE*. IEEE, 2019.

[10] CEA List. *Eclipse Formal Modeling Project*. https://projects.eclipse.org/projects/modeling.efm.

[11] S. Liu, J. Sun, Y. Liu, Y. Zhang, B. Wadhwa, J. S. Dong, and X. Wang. Automatic early defects detection in use case documents. In *ASE*. ACM, 2014.

[12] L. Lúcio, S. Rahman, C. Cheng, and A. Mavin. Just formal enough? automated analysis of EARS requirements. In *NFM*. Springer, 2017.

[13] A. Mavin, P. Wilkinson, and M. Novak. Easy Approach to Requirements Syntax (EARS). In *RE*. IEEE, 2009.

[14] S. Nogueira, H. Araujo, R. Araujo, J. Iyoda, and A. Sampaio. Test Case Generation, Selection and Coverage from Natural Language. *Sci. Comp. Prog.*, 2019.

[15] K. Pohl and C. Rupp. *Requirements Engineering Fundamentals - A Study Guide for the Certified Professional for Requirements Engineering Exam: Foundation Level - IREB compliant*. rockynook, 2011.

[16] S. Tripakis. Verifying Progress in Timed Systems. In *ARTS*. Springer, 1999.

[17] L. Wenbin, H. J. Huffman, and T. Mirosław. Temporal action language (TAL): A controlled language for consistency checking of natural language temporal requirements. In *NFM*, 2012.