Requirements Engineering for Hard Real-Time Systems: the Σ Notation and a Case Study

Marios Piveropoulos and Andy Wellings
Computer Science Department, University of York
York YO10 5DD, United Kingdom
{mario,andy}@minster.cs.york.ac.uk

January 1999

Contents

List of Figures

List of Tables

Abstract

Requirements engineering is generally acknowledged to be vital for every system development project, especially when the system to be developed has to meet hard real-time demands, usually while operating in a safety-critical environment. The Σ (Sigma) Notation, which is a new way of specifying formal well-structured requirements, suitable for hard real-time systems, is presented in this paper. The main characteristics of the notation are: formal underpinning, modelling flexibility, understandability from both engineers and users, rigorous handling of traceability for the derivation of requirements, and an interface to subsequent object-oriented analysis. We also present an extended example of the Σ Notation's use by modelling the Generalized Railroad Crossing.

1 Introduction

Requirements engineering is one of the major activities in a system's development lifecycle, and maybe the most important one. Mistakes introduced at this stage into the development are usually proven to be the most costly. In other words, it has been recognized that failure to identify the real application domain knowledge may result in poor quality, late delivery, and high maintenance costs. Furthermore, when the system itself is very complicated and important, such as a hard real-time system, this phase of the lifecycle becomes vital.

In order to overcome some of the problems of requirements engineering for hard real-time systems, the HRT-HOORA [1] method is currently being developed. This method uses the infrastructure of the already existing HOORA [7, 8, 9, 10], built for the needs of the European Space Agency. HRT-HOORA provides a formal, well-documented and well-structured way of eliciting, specifying, and tracing requirements, with special handling of real time constraints, along with producing a consistent preliminary analysis. It will also be closely coupled with the HRT-HOOD [1] design methodology, thus securing the coherency of a hard real-time development lifecycle up to the actual system implementation.

The Σ (Sigma) Notation was first conceived as an integral part of HRT-HOORA. The purpose of this paper is to introduce the Σ notation, define its semantics and tools, and give a sample of its use by means of modelling a well-known benchmark problem.

The major weaknesses of the HOORA method were found to be the lack of assistance to the analyst's interaction with the customer and the inability of expressing the requirements, at some stage of the analysis, in a formal way. HOORA's requirements model [8] is, all in all, a useful tool. However, in practice, it just provides for a way to group specific requirements under certain categories of requirement types. If the analyst uses only this model,

[1] Hard Real Time - Hierarchical Object Oriented Requirements Analysis.

he or she will have a difficult task clearly defining each requirement, translating them into a consistent object model and keeping track of requirements' derivation.

In order to accomplice all these tasks, we proposed the new Σ notation. The aim of this notation is to be used by the engineer as a first step towards the structured expression of user requirements. The input to a Σ model will be informally expressed user descriptions and the output formally expressed requirements structured in a uniform way.

All in all, the objective of this notation is to incorporate the following specific qualities:

- *Formality.* The notation should assist the formal expression of requirements, have formally specified semantics and a rigorous mathematical underpinning in order to produce unambiguous requirements models.

- *Verification of timing.* The notation is primarily aimed towards real-time systems and thus, verification of timing consistency in any Σ model should be ensured.

- *Expressivity and modelling flexibility.* The modelling of the requirements should be intuitive enough in order to enable better interaction between the stakeholders and the engineer, and therefore, assisting the *validation* of the model. The engineer should also be assisted with a highly flexible modelling technique.

- *Traceability handling.* The notation should be able to capture the derivation of requirements in a rigorous way, thus assisting a controlled evolution of the model and subsequent analysis and design.

The Σ notation is used as an integral part of HRT-HOORA, interacting with the rest of the components [2] as figure 1 depicts.

[2]The TRDTs, Lexicon and Requirements List are all HRT-HOORA integrated parts and are not discussed in this report.

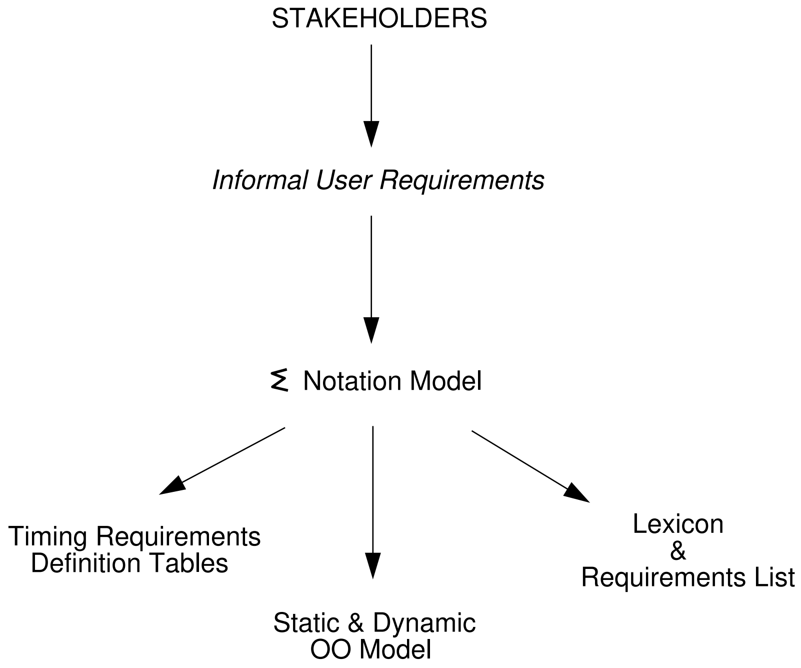

Figure 1: Input and output of the Σ Notation.

Most importantly, the engineer is able to map any Σ model onto an object-oriented model (static and dynamic), thus keeping the subsequent object-oriented analysis as consistent as possible with the requirements model. This direct relation between Σ notation and a subsequent object-oriented model makes it possible for the engineer to use the notation *outside the context* of HRT-HOORA, as a *precursor to object-oriented analysis.*

In the rest of this report we will briefly review the related work in the field (section 2), discuss in detail the semantics of the Σ notation with individual examples (section 3.1), introduce a mechanism for traceability of requirements (section 3.2), discuss verification, consistency and correctness of Σ models (section 3.3), give a rigorous interface to OO modelling (section 4), and present a Σ model for the well-known Generalized Railroad Crossing case study (section 5). Finally, conclusions and some present plans for further development are given in section 6.

2 Related work

Before focusing on the Σ notation itself, we first have to further justify its development and thus place it in the greater context of Requirements Engineering and system development. Therefore, we present a breef review of a few specification and development methods, namely VDM^{++} [6], Statecharts [11, 12], Real-Time UML [3] and ALBERT (more recently ALBERT II) [4, 5, 17]. These well-known methods arguably constitute a representative portion of today's cutting-edge practice in industry and academia. For extensive information on industry's practice on requirements engineering, the reader can have a look at the current results of the RESUME [3] project.

VDM^{++} has little to do with pure requirements engineering. It has, though, the capability for formal OO consistent designs for real-time systems, verification of attainability of system-level timing constraints and it can eventually reach up to prototype code generation. This is all bound together by the support of a tool set. Although this is a very useful set of capabilities it offers little or non-existant support for requirements engineering. In other words, there is no rigorous approach to explicit requirements handling, for example, support for requirements derivation. Also, the lack of a precursor to architectural design and the use of object-orientation from the very beginning of the development lifecycle can sometimes prove inflexible in expressing and structuring the specification.

Statechart offers a powerful modelling technique, by extending and improving the common state machines. It presents a useful alternative to structuring specifications and can prove very effective for defining reactive systems. It is also supported by the STATEMATE tool. Again, Statechart does not offer explicit and rigorous handling of requirements and can prove ineffective when specifying real-time systems because of the lack of the appropriate formalism. This is addressed by **Modecharts** [16] with the use of RTL, which is also used by the Σ notation as the underpinning real-time

[3] Project concludes in february 1999. Results at URL address: http://dsa-isis.jrc.it/SysReqs/resume/resume.html

formalism.

Real-Time UML is a diagramatic language, based on **UML** [4] [19, 20, 21, 22, 23]. That means that it employs standardized object-oriented concepts and notation for the effective modelling of complex systems. Apart from object-orientation, use cases, scenarios and stereotypes [5] are utilized and also statecharts for the behavioral modelling aspect. It also presents a coherent step-by-step development guidance up to detailed design. On the other hand, Real-Time UML does not really address requirements engineering in a rigorous way through the sole use of scenarios and use cases. Moreover, there is no formal underpinning for real-time development purposes, neither in the static aspect of the system (use cases, scenarios, OO model), nor in the dynamic (statecharts).

The **ALBERT** (and **ALBERT II**) language aims primarily at requirements engineering. It introduces an elaborate notation and methodology for coherent real-time modelling with explicit handling of requirements. Because, of the close coupling of informal and formal requirements, as in the Σ notation, validation of the model is inherently much easier. On the other hand, there is no capability of handling the derivation of requirements and keeping requirements' traceability.

The Σ notation strives to cover *in one integrated method* most of the shortcomings of the above methods by:

i. offering a flexible structuring technique with a mathematical underpinning,

ii. supporting formal and consistent real-time modelling,

iii. providing rigorous handling of requirements and requirements' traceability,

iv. establishing structural consistency and correctness of the model,

[4] Unified Modeling Language.

[5] Meta-classification of modelling elements.

v. enabling algorithmic translation into object-orientation with capability for subsequent model-checking for further verification of timing,

vi. being part of HRT-HOORA which provides a complete modelling and working framework especially for large projects and

vii. having a strong connection (through HRT-HOORA) to HRT-HOOD which offers a detailed design, timing schedulability analysis and code generation.

3 The Σ Notation

3.1 Semantics in detail

In the following sections we will discuss the semantics of the Σ notation in detail. A string of simple examples will be presented in order to enhance the understanding of the notation. For that matter, the following user description of a simple hypothetical component will be used.

> We have a certain embedded component C which consists of a sensor S and an actuator A. When there is a certain change c_i in the environment, component C must detect it and subsequently produce a certain action a_i. The action must occur within t mseconds of the occurance of the change.

3.1.1 The 'system' notion

The basic concept in the Σ notation is the SYSTEM, marked with the letter Σ [6]. In most of the cases a Σ-system will be a real-world entity, but it is left at the disposal of the engineer to view as a Σ-system anything that would assist the model specification and the succeeding analysis.

The Σ-system is the basic structural building block of every Σ specification. The main function of the notation is to formally specify models

[6] The letter Σ is the initial of the greek word for 'system'.

of embedded systems by using aggregation relationships among different Σ-systems. So, a single Σ-system can include other Σ-systems and/or have sibling Σ-systems [7]. The engineer can formally specify the interaction through signals among different systems, give them certain attributes and functionality and limit them with certain constraints.

At this point we must also note the use of the special system **ENV** in Σ models. This is used when an outside environment for a certain system is not stated in the primary sources or it is of no particular interest to us. Moreover, in large models, where we might have more than one unspecified environments that must be declared or we just want to emphasize the existance of a single environment for many different systems, multiple **ENV** Σ-systems can be declared.

In such a case, single or combinations of capital latin letters or even whole words can be used as subscript labels, for example:

$\mathbf{ENV}_{A}$, $\mathbf{ENV}_{B}$, $\mathbf{ENV}_{A1}$, $\mathbf{ENV}_{GENERAL}$

Collectively, every Σ-system can be specified by the following parts:

- Header *(mandatory)*
- Local Variables
- Interaction *(mandatory)*
- Logic *(mandatory)*
- Constraints

The formalism of a Σ specification lies primarily (but not only) in the use of the event-action model introduced in [15], and the Real Time Logic (RTL). Model specification is also assisted by the use of algebraic expressions, arithmetical relations and first order predicate logic.

[7] From this point forward Σ-systems can also be referred to as simply systems.

3.1.2 The header

The header, written always in boldface, is the first part of every Σ-system specification. It is an important part since it specifies the Σ-system's name and its place in the inclusion hierarchy in a certain model.

Let us consider the set SYSTEM which includes all the possible Σ-systems. So, if i, j : SYSTEM and K== $\{n$:SYSTEM $\mid n \notin \{$i, j, **ENV**$\}\}$, then the header must be written according to the following template:

[**i**] Σ**:** **j** [**K**]

This means that the specification to follow is of the system j, which is included (**aggregation relationship**) in system i, and that system j includes a set K of other systems. Set K can be an empty set.

EXAMPLES:

[**ENV**] Σ**:** *C* [*S*, *A*]
(0,0,-):/We have a certain embedded Component C which consists of a Sensor S and an Actuator A/ [8]

[*C*] Σ**:** *S* [ ] *(0,0)*

[*C*] Σ**:** *A* [ ] *(0,0)*

3.1.3 Local variables

This is the second part of system specification but it can be omitted if the engineer has no use for it in a particular system. As the name suggests, the engineer can declare variables and their types for a certain system, using

[8]The purpose of natural language sentences in a Σ model and the information captured in the preceeding brackets will be explained in section 3.2.

standard mathematical notation. The propositions that are formed are automatically set in logical conjunction with each other.

EXAMPLES:
These could be local variables to Σ: C

Local Variables
in_signals : $\aleph$
env_change : $\{c_1, \ldots, c_n\}$
state : {idle, processing}

3.1.4 Interaction

In this part all possible interactions or *signal sending* between the Σ-system in question and its environment must be declared. The environment of a Σ-system is:

*a)*the Σ-system which includes it, and

*b)*other 'sibling' Σ-systems

In order to achieve this, we use formal statements which must follow a certain template. The formal statements are thought to be automatically in conjunction, unless disjunction is otherwise specified at places [9].

So, if i, j : SYSTEM and a : ACTION [10] , the following template is formed:

i(S: [a]) $\triangleright$ j

[9] That would be a highly unlikely occurrence because we would then specify alternative ways of the system's communication with its environment which would constitute grounds for the existance of more than one systems. Nevertheless, this freedom of multiple system interfaces is given to the engineer as an alternative modelling dimension.

[10] A primitive action is a schedulable unit of work which consumes a bounded amount of system resources (cpu or communication bandwith) according to [15]. A composite action is a partially ordered group of other (primitive) actions.

This means that system i can interact with system j through signal S:[a]. The direction of the signal is shown by the small triangle; so in this case system i can potentially send the particular signal *to* system j. The letter 'a' is **in square brackets** because we want to show that it **is an action**. So, collectively, *system i can send to system j the signal to begin (composite) action [a].* We can also have *control signals* e.g. (S: OK), which do not cause the receiver to begin a new action but can change its state.

At this point we should make the following observations:

- (S: [a]) is an *event* (signal event)
- If we know that a distributed solution will not be implemented or we simply consider the time that the signal takes to reach its destination as negligible, then we can define the following axiom [11]:

$$\boldsymbol{\forall i : @((S:\ [a]), i) = @(\uparrow [a], i) \Rightarrow}$$
$$\boldsymbol{(S:\ [a]) \equiv\ \uparrow [a]}$$

  In other words the beginning of a certain action $[a]$ and the event $(S:\ [a])$ which motivates it, are considered to happen instantly and at the same time. Therefore, $(S:\ [a])$ and $\uparrow [a]$ can potentially be interchangeable events, especially in expressions of timing constraints that we will examine later on in this report.

EXAMPLES:
This is the interaction part of the Σ: C

Interaction

(0,1,-):/When there is a certain change in the environment the Component must detect it/
ENV(S: $[c_i]$) $\triangleright$ C

[11]Where $@(e,i)$ is the RTL occurence function ([15]).

(0,2,-):/The Component will notify its environment when the action is completed/
C(S: ActionCompleted) ▷ ENV

and this would be the interaction part of the Σ: S

Interaction

(1,1,(0,1)):/The Sensor will be informed by the Component of any change in the environment/
ENV(S: $[c_i]$) ▷ C(S: $[c_i]$) ▷ S

(1,0,[(0,1),(0,2)]):/The Actuator will be informed by the Sensor of any change in the environment/
S(S: $[react_i]$) ▷ A

3.1.5 System plugs

Let us assume that i, j, k : SYSTEM and that j is included in i, and k is included in j. Hypothetically, we could have a case where a signal had to be handled like this:

i(S: [a]) ▷ j(S: [a]) ▷ k

This is, actually, a common situation when using the Σ notation; a system k has to receive a signal from a system i which is not in k's immediate environment (which in this case is sytem j).

To write the whole path of the signal can become quite cumbersome ,especially when the number of the included systems increases. So, although this way can be very helpful when trying to trace the signal back to its source, an alternative way must be provided.

This is the case where the system **plugs** can be utilised. Every system can have certain plugs through which it can communicate with its environment. Each plug is connected to a certain signal either *a)coming from* the outside environment (we use a **plug-in**) or *b)going to* the outside environment (we use a **plug-out**). Each plug can inherit the name of the signal that it is connected to, or carry any other meaningful name chosen by the engineer.

So, the previous example would be handled like this:

(in the interaction part of system j)

i(S: [a]) $\triangleright$ j(plug-in: [a])

(in the interaction part of system k)

j(plug-in: [a]) $\triangleright$ k

The system plugs also facilitate the *reuse* of selected systems by strengthening the system's 'black box' perspective of its outside world. Therefore, when a system has a certain functionality and properties which can potentially be used in different environments, the engineer can just unplug the system's specification from its previous environment and plug it in its new one.

3.1.6 Logic

The purpose of this part is to specify the behaviour of the system by formally breaking down each composite action ([15]). This produces further (primitive) actions and/or the sending of signals, which can also be seen as primitive actions with negligible execution time [12]. We complete this part by specifying expressions using the special symbol $\rightharpoonup$, which has the meaning of *trigger or release*. First order predicate logic can also be used, and in fact the formal expressions are placed automatically in conjunction unless otherwise specified.

[12]In a distributed system, the sending of signals would be modelled as primitive actions but with accountable durations.

So, if a, b, c : ACTION, then we could have the following very simple case:

$\uparrow$[a] $\rightharpoonup$ [b];[c]

This means that *the event of the beginning of composite action* [a] *triggers or releases the execution of the primitive action* [b] *followed by the execution of the primitive action* [c]. This can also be seen as the decomposition of composite action [a] into a temporally ordered group of primitive actions.

For the sake of briefness we would usually write:

[a] $\rightharpoonup$ [b];[c]

These expressions can be seen as having a *dual formal underpinning*, since we can describe them not only by the event-action model in [15] but also as causal relationships. The causal model that we use for our purposes is defined in [18].

Therefore, if *effect* is a temporally ordered group of primitive actions, then we define that the event $\uparrow$[a] **is a necessary and sufficient direct cause** of *effect* as defined in [18]. This means that, in all possible observation histories of a particular system, whenever $\uparrow$[a] occurs it is always followed [13] by $\uparrow$*effect*. So, temporally speaking, we have that:

$$\forall i : @(\uparrow [a], i) \leq \ @(\uparrow \mathit{effect}, i)$$

In order to avoid non-determinism, since we are interested in *hard* real-time, we ammend this definition by specifying that in a Σ model we also have:

$$\forall i : @(\downarrow \mathit{effect}, i) = \ @(\downarrow [a], i)$$

In other words, we specify that the completion of the caused effect is set to coincide with the completion of the triggering action.

[13] The task (or thread) that will eventually correspond to *effect* becomes runnable but possibly does not execute immediately due to inteference, blocking or release jitter.

We could also have the following:

[a] $\rightharpoonup$ [b];[(S: [c])]

This means that the event of the beginning of action [a] causes action [b] which is followed by the sending of the signal S: [c]. The receiver of the signal should be stated in the interaction section.

Note that state transitions [14] caused by control signals or not, can be modelled as primitive actions (with negligible execution time), like the following:

(S: Takeoff) $\rightharpoonup$ [set(status, flying)]

EXAMPLES:
This would be the Logic part of the Σ: S

Logic

(2,0,(1,0)):/When a change occurs the Sensor will inform the Actuator with a specific message/
[c_i] $\rightharpoonup$ [(S: [$react_i$])]

and this would be the Logic part of the Σ: A

Logic

(1,2,(0,2)):/The Actuator will carry out a certain action and then notify the Component that the action is completed/
[$react_i$] $\rightharpoonup$ [a_i];[(S: ActionCompleted)]

[14]These are seen as events in the event-action model introduced in [15].

3.1.7 Constraints

The purpose of this last part is to enable the engineer to formally specify any particular constraints (duration constraints upon actions mainly), which, the system at hand, must satisfy at all times. The tools here are the usual mathematical notation, first order predicate logic and RTL.

EXAMPLE:
This would be the constraints part of the Σ: C

Constraints

(0,3,-):/The action must occur within t mseconds of the occurance of the change/
$\forall j : 0 < @(\uparrow[a_i], j) - @(\uparrow[c_i], j) \leq t\ ms$

3.2 On the trail of requirements

In order to assist the communication between the engineer and the various project stakeholders, the Σ notation provides coupling of informal and formal requirements. Thus, unless it is decided otherwise for strictly individual cases, each formal statement found in a Σ model must be coupled with the informally expressed requirement (typically in plain text) that it wholly or partially represents. These informal sentences, in most cases, can not be found in the original requirements, since they are derived from them.

As a result of this practice, the engineering team is liable to get a much better feedback from the appropriate project stakeholders and so, the validation and completeness checking of the requirements model becomes quicker, easier and more reliable.

Moreover, an integral part of the Σ notation is the facilitation of *requirements traceability* during the system analysis lifecycle, which can be extended for the entire project lifecycle.

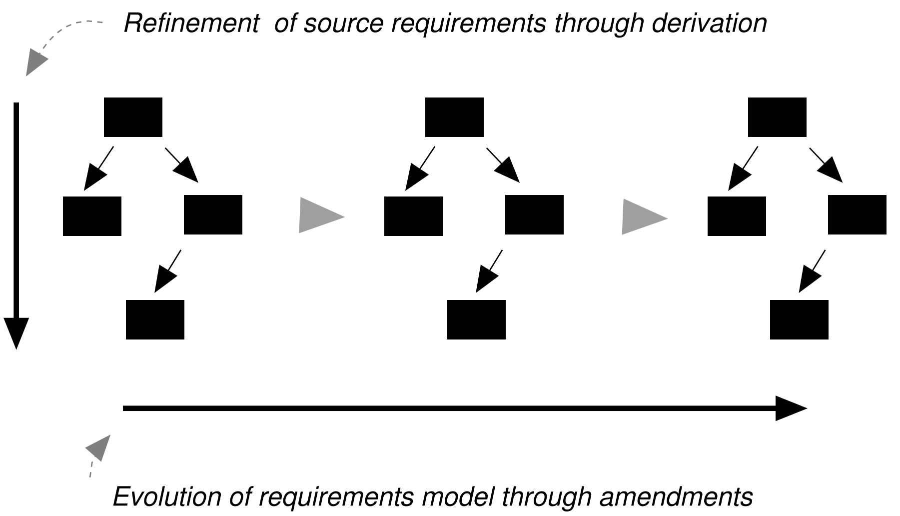

Figure 2: Requirements' derivation and evolution.

Due to various design decisions being made in requirements specifications ([2]), it is established that original user requirements are refined during project development through *derivation*. Moreover, the requirements model *evolves* through constant change and various improvements (see figure 2). Thus, in order to keep the consistency and coherency of an ever evolving and changing requirements model, we need to be able to rigorously trace individual requirements back to their sources.

In order to establish a common view point on this issue, we need to define what requirements derivation is perceived to be, in the following text. Thus, the technique followed by the Σ notation states that a derived requirement is usually more detailed or less abstract than its sources, although this is not compulsory. Most importantly though, *fulfillment of an arbitrary source requirement at any level implies the compulsory fulfillment of all of its derived requirements*; that does not mean though, that fulfillment of the derivatives leads to automatic fulfillment of the source. The requirements engineer may decide not to follow this protocol, and derive requirements in a different manner. In that case, the alternative protocol should be made clear for inspection to the appropriate project stakeholders (end users, other engineers, etc.).

So, unless otherwise decided by the engineer in individual cases, each formal statement that belongs to a Σ model should be coupled with an *ordered* triplet (x, y, I), where

$$x, y : \aleph \ \wedge \ I == \{i, j : \aleph \bullet (i, j)\}$$

Following are the semantics and rules for each part of the triplet:

- The first part (x), is a natural number indicating the *level of derivation.* At level 0 we have all the original user requirements, perhaps with minor changes to assist the engineer's work. At subsequent levels (1...n) we have derived, usually more specialized, requirements, possibly expressed by the engineer, and based on requirements already documented at previous levels. So, for example, a certain requirement at level 2 is derived from one or more requirements from level 1 and perhaps, level 0, and can be used to derive one or more requirements at level 3 up to level n. Ultimately, all requirements are based on those at level 0, the original user requirements. If we wanted to depict this network of requirements, it would show as multiple interelated trees, the roots of which can not exceed the number of the original user requirements (see figure 3).

- The second part (y), is a natural number that uniquely marks a requirement in a certain level. So, the *global key* for a specific requirement would be the *ordered pair* (x, y) of the first and second integers.

- The third part is a set [15] of global keys specifying requirements at previous levels, from which the present requirement was *directly* derived. Therefore, every requirement has a full record of its 'parent' requirements, and so, it can be traced back recursively to its original sources.

  Note that in a triplet (x, y, I) we have that:

$$x = k \Longrightarrow \exists (i, j) \in I \mid i = k - 1$$

[15] There is at least one, except when we specify a requirement at level 0. In this case the third part is an empty set.

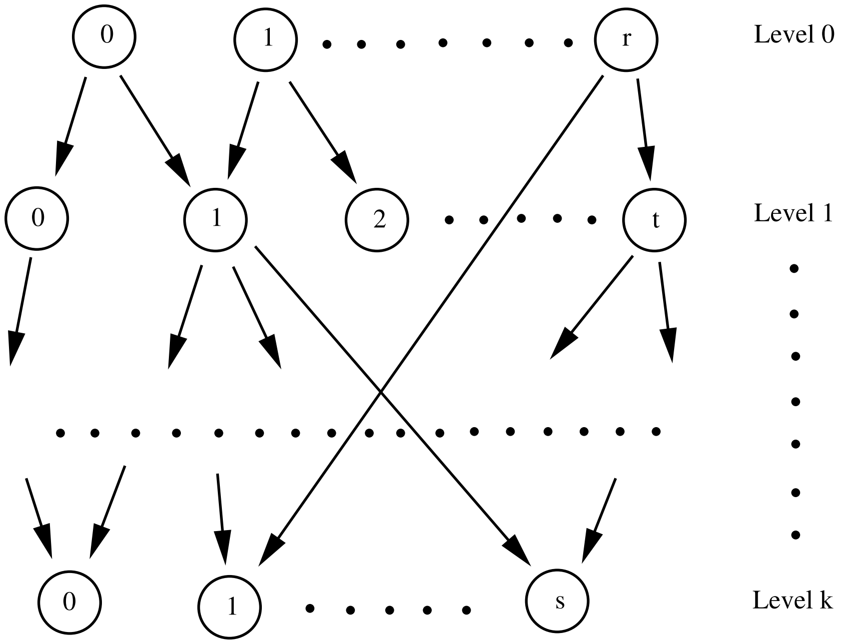

Figure 3: A requirements' derivation network.

which means that if the present requirement is on a certain level, then *at least one* of its 'parent' requirements must be on the immediately preceding derivation level.

This third part is depicted only at the first occurence of a certain requirement in a Σ notation model. At this first occurence (or wherever the engineer deems necessary) the actual text of the requirement is also presented in natural language, using *emphasized* fonts [16]. This helps to make the link between the informal primary sources and the subsequent formally stated requirements. On other instances, when the same requirement needs to be refered to, only the global key is used, which is sufficient to specify it.

[16]Emphasized text can also be used for comments.

3.3 Consistency issues: the extended Σ notation

3.3.1 Aim and definitions

The Σ notation, as it was shown in previous sections, is rigorously defined and is based on formal concepts (event-action model, RTL, signalling). In the form that it was presented it can be used to capture and structure user information and define unambiguous requirements with traceability guidance.

In this section we discuss the second part of the notation which can be used by the engineer in order to reason about the information captured in a Σ model. That formalism we will call the *extended* Σ notation. In fact, the extended notation is a mathematical approach which aims to aid the verification of timing consistency among Σ-systems and other consistency-related aspects in a Σ model. This implies two things:

- The extended notation does not specify requirements but helps to reason about them
- The extended notation uses an already defined Σ model as a knowledge base

The extended notation is based on the fact that any Σ-system can be seen as a structured collection of requirements or formal sentences (there is a 1 to 1 relation between the two as we have seen) or, to put it in a more mathematical way, a *set of formal propositions*.

The basic tool of the notation is the function $\sigma()$. The $\sigma()$ function is *generic*, which means that each individual Σ-system from a certain model utilises its own 'privatized' instance of the $\sigma()$ function. The 'privatized' $\sigma()$ function is denoted by indexing, according to the following template:

$$\sigma_{SystemName}()$$

The $\sigma()$ function is also *overloaded*, meaning that it stands for a whole family of specialized functions, and therefore, it can handle a changing set of parameters. Hence, given that I is such a set of acceptable parameters, we define that:

$$I \in \mathbf{P}\{p, C, \vartheta_1, \vartheta_2, \vartheta_3, \vartheta_4\}$$ [17]

Therefore, the engineer can select a subset of the six standardized arguments given above, each of which has a special meaning or *correspondence.* According to the arguments used, the function *will produce a set* containing specific formal propositions and/or other information defined in the designated Σ-system:

$$\sigma_{SystemName}(I) = \{\ldots\}$$

The cardinality of the set produced will be equal to the number of the parameters used in the function. In fact, there is a 1 to 1 relation between each of the six possible parameters $p, C, \vartheta_1, \vartheta_2, \vartheta_3, \vartheta_4$ and a member of the set that will be returned by the function. In order to define the exact correspondence we need to identify the critical parts of a Σ-system and label them. This is done in the following template:

[system-P] Σ: SystemName [set-C]
Local Variables
proposition-α
Interaction
proposition-β
Logic
proposition-γ
Constraints
proposition-δ

We can infer, using the definition of the semantics of Σ-systems, the following:

- Σ: system-P is the system that includes Σ: SystemName
- Σ: SystemName includes other Σ-systems which can be found in set-C

[17] The $\vartheta_{1\ldots4}$ symbolism was selected in order to denote partiality. This will become obvious later in this text.

- Proposition-α, proposition-β and proposition-γ can be seen as purely conjuctive propositions (unless otherwise specified at places) made out of formal sentences. Also, proposition-δ is just a more freely binded (not only conjuctive) logical proposition made out of, mainly, RTL expressions.

Having that general template in mind, we can now show the mapping of the six possible arguments in a $\sigma()$ function, to information captured in a Σ-system. This is shown in table 1.

Argument mapping	
Argument	Σ-system part
p	system-P
C	set-C
ϑ_1	proposition-α
ϑ_2	proposition-β
ϑ_3	proposition-γ
ϑ_4	proposition-δ

Table 1: Mapping arguments to Σ-system parts.

Thus, application of the $\sigma()$ function would produce the following values:

- $\sigma_{SystemName}(p) = \{$system-P$\}$.
- $\sigma_{SystemName}(C) = \{$set-C$\}$.
- $\sigma_{SystemName}(\vartheta_1) = \{$proposition-$\alpha\}$.
- $\sigma_{SystemName}(\vartheta_2) = \{$proposition-$\beta\}$.
- $\sigma_{SystemName}(\vartheta_3) = \{$proposition-$\gamma\}$.
- $\sigma_{SystemName}(\vartheta_4) = \{$proposition-$\delta\}$.

- $\sigma_{SystemName}(p, C, \vartheta_1, \vartheta_2, \vartheta_3, \vartheta_4)$ = {system-P, set-C, proposition-α, proposition-β, proposition-γ, proposition-δ}. The value of the function includes all the requirements defined in the Σ-system. Thus, we define that:

$$\sigma_{SystemName}(p, C, \vartheta_1, \vartheta_2, \vartheta_3, \vartheta_4) = \Sigma\text{: SystemName}$$

As a notational convention, we also define that:

$$\sigma_{SystemName}(p, C, \vartheta_1, \vartheta_2, \vartheta_3, \vartheta_4) = \sigma_{SystemName}()$$

- Some examples of using alternative set of arguments are the following:
  - $\sigma_{SystemName}(p, \vartheta_4)$ = {system-P, proposition-δ}.
  - $\sigma_{SystemName}(p, C)$ = {system-P, set-C}.
  - $\sigma_{SystemName}(C, \vartheta_2, \vartheta_3)$ = {set-C, proposition-β, proposition-γ}.
  - $\sigma_{SystemName}(p, \vartheta_1, \vartheta_2, \vartheta_3, \vartheta_4)$ = {system-P, proposition-α, proposition-β, proposition-γ, proposition-δ}.

Since the $\sigma()$ function uses sets to represent Σ-systems, the usual set operations can be applied in the notation.

The $\sigma()$ function is what connects a standard Σ model, or the knowledge base, with the extended notation, or the mathematical formalism. In addition, the extended notation offers two more functions which can be used in conjunction with the $\sigma()$ function.

The first of these additional, but nevertheless very important functions, is the deset() function. If SET is the set of all possible sets then the definition of the function is the following:

$$deset == \{\forall\{x\} : SET \bullet \{x\} \mapsto x\}$$

More descriptively, this would mean that the function accepts as argument a set containing a single member and returns the member by itself. This can be a very useful operation in the extended notation because we can express interesting propositions such as in the following generalized example:

$$\neg(deset(\sigma_{systemA}(\vartheta_4)) \wedge deset(\sigma_{systemB}(\vartheta_4))) = F$$

According to the example, the negation of the conjunction of the constraints of *systemA* and *systemB* is false.

Function mkdscr() [18] is also an important addition to the extended notation. Assuming that PROP is the set of all possible propositions and CONN is the set of all logical connectives, we would define the function as follows:

$$mkdscr == \{\forall x_1 \ldots x_n : PROP; c_1 \ldots c_{n-1} :$$
$$CONN \bullet \{x_1 c_1 x_2 c_2 \ldots c_{n-1} x_n\} \mapsto \{x_1, x_2, \ldots, x_n\}\}$$

In other words, the function accepts as arguments sets which contain complex expressions of logically connected propositions, and returns a set containing the propositions as separate members, thus 'forgetting' the connective information. An important observation on this functionality is that it is *irreversible*, meaning that a product of this function can not be switched back into its previous state. Nevertheless, this function is quite important because we can represent useful reasoning in the following manner:

$$i(S : [a]) \triangleright j \quad \in \quad mkdscr(\sigma_j(\vartheta_2))$$

That means that the formal sentence $i(S : [a]) \triangleright j$ can be found in the interaction definition of Σ-system j.

3.3.2 Guidelines for consistency

Consistency of timing in a Σ model of a hard real-time system, is obviously of the essence; and although a Σ model can produce a dynamic OO model that, in a latter stage, can be rigorously model-checked with respect to its timing properties, we have to ensure that a minimum of timing consistency exists outright in any Σ model. In order to prove minimal timing consistency in a Σ model, we utilise the extended notation.

[18] The name comes from the words *make discrete*.

Minimal timing consistency can be described as the property of a Σ model by which different Σ-systems in the model do not carry timing constraint definitions that can contradict each other. Conflicting timing constraints would result into an apparently non-uniform view of time in the model. Therefore, we have to impose relational restrictions among Σ-systems.

Because of the hierarchical nature of a Σ model, due to system inclusion, we need primarily to ensure the consistency between parent and children [19] Σ-systems. Undetected inconsistency in parent-children constraints would possibly lead to schedulability problems for the latter design, mainly being, unschedulable threads, resulting to, for example, end-to-end deadlines that cannot be met.

$deset(\sigma_x(\vartheta_4))$	$\Rightarrow$	$deset(\sigma_{y_1}(\vartheta_4))$	$\wedge$	$deset(\sigma_{y_2}(\vartheta_4))$
t	T	t	T	t
t	F	t	F	f
t	F	f	F	t
t	F	f	F	f
f	T	t	T	t
f	T	t	F	f
f	T	f	F	t
f	T	f	F	f

Table 2: This truth table shows the timing constraint dependency between one system and its two subsystems. In a valid, from a timing point of view, Σ model, this implication relationship must always be true.

So, minimal timing consistency is ensured in a parent-children relation when, $\forall x, y_1, \ldots, y_n$: SYSTEM | $deset(\sigma_x(C)) = \{y_1, \ldots, y_n\}$, the following proposition is always true:

$$deset(\sigma_x(\vartheta_4)) \Rightarrow deset(\sigma_{y_1}(\vartheta_4)) \wedge \ldots \wedge deset(\sigma_{y_n}(\vartheta_4))$$

[19]The terminology *parent* and *children* is used in order to denote a hierarchical status of Σ-systems via inclusion (aggregation or decomposition).

Table 2	$\Leftrightarrow$	$\neg deset(\sigma_x(\vartheta_4))$	$\Leftarrow$	$\neg deset(\sigma_{y_1}(\vartheta_4))$	$\vee$	$\neg deset(\sigma_{y_2}(\vartheta_4))$
T	**T**	f	T	f	F	f
F	**T**	f	F	f	T	t
F	**T**	f	F	t	T	f
F	**T**	f	F	t	T	t
T	**T**	t	T	f	F	f
T	**T**	t	T	f	T	t
T	**T**	t	T	t	T	f
T	**T**	t	T	t	T	t

Table 3: Contraposition of the primary implication as means of proof. De Morgan's law is applied to the antecedent part.

That tells us that the conjunction of the children's timing constraints is implied by the timing constraints of the parent system and thus securing that the children cannot contradict the parent. In other words, it is impossible for the constraints of the parent to be effective without those of its children being true as well. This is shown in table 2, simplified for the case of a parent having only two children.

In order to prove that this primary implication is true for a parent-children relation in a particular Σ model, we can instead show its contraposition; that the negated conjunction of the children's constraints implies the negated constraints of the parent or, in other words, a set of constraints that cannot stand true while our original set of parent constraints is considered true as well.

$$\neg(deset(\sigma_{y_1}(\vartheta_4)) \wedge \ldots \wedge deset(\sigma_{y_n}(\vartheta_4))) \Rightarrow \neg deset(\sigma_x(\vartheta_4))$$

This is a quite useful approach because by performing calculations on the antecedent part of the contraposition, we just need to reach a single expression that cannot have the same truth value as the parent's constraints. This equivalence is shown in table 3, again for the case of two children.

At this point we must also note that by convention we have:

$$\sigma_{SystemName}(\vartheta_4) = \{\} \Rightarrow deset(\sigma_{SystemName}(\vartheta_4)) = T$$

In other words, non-existant constraints are always considered to be true.

In order to have timing consistency we must make sure that our model is coherent in a general sense as well. Therefore, we need to follow a number of rules which will ensure structural consistency and more generally, the correctness of a model written in the standard Σ notation.

At this point, we reiterate that SYSTEM is an infinite set containing all possible Σ-systems, and ACTION is an infinite set containing all possible actions. Furthermore, MODEL is a finite set containing the Σ-systems in a specific model.

We also assume that **i)**a : ACTION, **ii)**i, w, x, y, k : SYSTEM, **iii)**$w, x, y, k \in$ MODEL, **iv)**$k \in deset(\sigma_x(C))$ and **v)**$x, y \in deset(\sigma_w(C))$. The hierarchy resulting from **(iv)** and **(v)**, which covers all possible cases, is visualized in figure 4. The rules are:

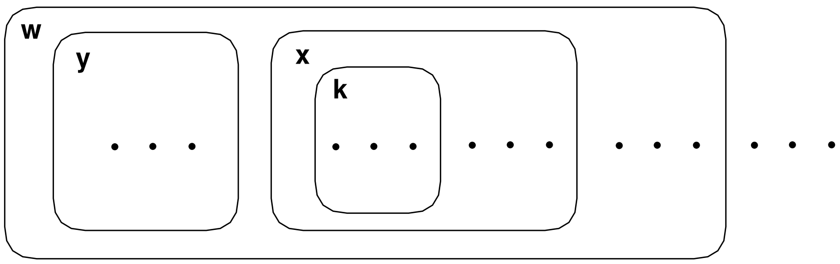

Figure 4: Example hierarchy.

1. $\forall i \Rightarrow \mid \sigma_i(p) \mid = 1 \ \wedge \ \mid \sigma_i(C) \mid = 1 \ \wedge \ \mid mkdscr(\sigma_i(\vartheta_2)) \mid \ > 0 \ \wedge$ $\mid mkdscr(\sigma_i(\vartheta_3)) \mid \ > 0$

   Every Σ-system must have a specified parent, set of children (which can be empty), interaction part and Logic part.

2. $x(S : [a]) \triangleright y \ \in \ mkdscr(\sigma_x(\vartheta_2))$

   if and only if

   $x(S : [a]) \triangleright y \ \in \ mkdscr(\sigma_y(\vartheta_2))$

   Sibling Σ-systems should both contain their mutual interaction.

3. (a) **If** $k(S:[a]) \triangleright x \;\in\; mkdscr(\sigma_k(\vartheta_2))$

   then $k(S:[a]) \triangleright x \;\notin\; mkdscr(\sigma_x(\vartheta_2))$

   (b) **If** $x(S:[a]) \triangleright k \;\in\; mkdscr(\sigma_k(\vartheta_2))$

   then $x(S:[a]) \triangleright k \;\notin\; mkdscr(\sigma_x(\vartheta_2))$

   A Σ-system should contain the interaction definitions with its parent, but the parent does not contain interaction definitions with its children.

4. (a) $k(S:[a]) \triangleright x(plug-out:[a]) \;\in\; mkdscr(\sigma_k(\vartheta_2))$

   If and only if

   $\exists y:\; x(plug-out:[a]) \triangleright y \;\in\; mkdscr(\sigma_y(\vartheta_2))$

   $\vee$

   $\exists w:\; x(plug-out:[a]) \triangleright w \;\in\; mkdscr(\sigma_x(\vartheta_2))$

   (b) $y(S:[a]) \triangleright x(plug-in:[a]) \;\in\; mkdscr(\sigma_x(\vartheta_2))$

   $\vee$

   $w(S:[a]) \triangleright x(plug-in:[a]) \;\in\; mkdscr(\sigma_x(\vartheta_2))$

   If and only if

   $\exists k:\; x(plug-in:[a]) \triangleright k \;\in\; mkdscr(\sigma_k(\vartheta_2))$

   The connection of system plugs is ensured. All plugs must be connected either terminally to a Σ-system or to another plug, by extension of the rule (see note at the end of these rules).

5. $[a] \rightharpoonup\; primitiveactions \;\in\; mkdscr(\sigma_x(\vartheta_3))$

   $\vee$

   $[a] \wedge condition \rightharpoonup\; primitiveactions \;\in\; mkdscr(\sigma_x(\vartheta_3))$

   if and only if

   $y(S:[a]) \triangleright x \;\in\; mkdscr(\sigma_x(\vartheta_2))$

   Here *primitiveactions* is one or more primitive actions in a certain succession (see semantics of the Logic system part in section 3.1.6), and *condition* is a logical expression that serves as an additional guard to the triggering of the primitive actions.

That ensures all composite actions assigned to a specific Σ-system are accounted for in the Logic of the same system.

Note that rules 2, 3, 4 and 5 *can be extended* further *to include the use of system plugs*, where they are not already specified. In that case, the engineer has to ensure that the appropriate semantics, as explained in section 3.1.5, are not violated.

An example of rule extension by using system plugs is the following:

Extended rule *4(a)*:

$k(plug - out : [a]) \triangleright x(plug - out : [a]) \;\in\; mkdscr(\sigma_k(\vartheta_2))$

If and only if

$\exists y :\; x(plug - out : [a]) \triangleright y(plug - in : [a]) \;\in\; mkdscr(\sigma_y(\vartheta_2))$

$\vee$

$\exists w :\; x(plug - out : [a]) \triangleright w(plug - out : [a]) \;\in\; mkdscr(\sigma_x(\vartheta_2))$

4 Semi-mechanized transformation to OO modelling

4.1 Primary interface

Working in the context of HRT-HOORA (and not only), a completed Σ model will be translated into its corresponding static and dynamic object-oriented models.

The complete interface to an OO model can be defined as follows (in pseudocode):

begin translation
 define smodel == static OO model;
 define appended_smodel == static OO model appended with state variables

define dmodel_{p1} == partially defined dynamic OO model(CIDs);
define dmodel_{p2} == partially defined dynamic OO model(STDs);
define dmodel == dynamic OO model;

$\text{algorithm}_\alpha(\Sigma \text{ model}) = \{\text{smodel}, \text{dmodel}_{p1}\}$;
$\text{algorithm}_\beta(\Sigma \text{ model, smodel}) = \{\text{dmodel}_{p2}, \text{appended\_smodel}\}$;
$\text{dmodel} = \text{dmodel}_{p1} + \text{dmodel}_{p2}$;
$\text{OOmodel} = \text{appended\_smodel} + \text{dmodel}$;
end translation

The two algorithms integrated as functions in this transformation to OO modelling are specified in sections 4.2 and 4.3. It is easily observable that the primary interface just presented, is mechanized [20]. On the other hand, the two algorithms rely for their completion on human intervention and decision making at certain points, and thus, we refer to the entire transformation as *semi*-mechanized.

Basic characteristics and concepts of HOORA's OO model are inherited by HRT-HOORA. Thus, terminology used by HOORA, and therefore unambiguously defined in HOORA's reference manual [8], will be *emphasized* in most occurrences through the following algorithm definitions.

4.2 Algorithm$_\alpha$

1. A HOORA *external class* must be contructed for each $\textbf{ENV}_{label}$ special Σ-system.

2. Consider the highest level of abstraction in the Σ model.

3. Take *one and only one* of the following substeps:

   (a) Consider one Σ-system, which has not already been examined, in the current level of abstraction and construct one HOORA *class*,

[20] So, consequently it could be fully automated by a computing tool.

optionally with the same name. For the purpose of this algorithm we will refer to them as the **target system** and the **target class**.

(b) Consider one Σ-system, which has not already been examined, in the current level of abstraction and construct more than one HOORA *classes*. The number of such classes will be in agreement with the semantics of the Σ-system and strategic decisions of the engineer.

The rest of this algorithm's steps will be taken once for each of the classes that are constructed. Each time the **target system** will be the Σ-system and the **target class** one of the classes produced.

In this case, the functional and non-functional requirements in the Σ-system are 'inherited by more than one HOORA classes. On the other hand, the engineer is able to specialize certain classes more than the others by altering slightly their details [21].

(c) Consider more than one Σ-systems, which have not already been examined, in the current level of abstraction and construct one HOORA *class*. In this case, the Σ-systems must be *strongly conceptually and semantically related* [22]. Variables, functionality, and constraints will be merged as if they were parts of larger, *super* Σ-system [23].

For the purpose of this algorithm we will refer to the Σ-systems collectively as the **target system** and to the class as the **target class**.

(d) Consider one Σ-system, which has not already been examined, in the current level of abstraction and construct an *association* relationship between its 'parent' Σ-system and another system in

[21] Appointing less *operations* to a certain class, for example.

[22] For example, an array of sensors, that the engineer wants to model as one class, but are already modelled as a number Σ-systems.

[23] Which means that structural and correctness rules must be enforced for the merged system.

the parent's environment. In this case the functionality of the Σ-system passes over to its 'parent' [24] system, together with its local variables and constraints. It is good practice though, not to attempt such a mapping when there are important non-functional requirements (timing constraints) present in the system.

If the present substep is taken, the algorithm must be *re-run* for the specific parent Σ-system, thus taking into account the additions made to it.

4. Each local variable of the target system will be an *attribute* of the target class, carrying over the same type and general semantics.

   The engineer can add new supplementary *attributes* if he or she deems necessary.

5. For each incoming Σ-signal to the target system, a corresponding HOORA *operation* must be defined for the target class. If the signal is received by a plug-in in the target system, then the corresponding *operation* is *covered* by an *operation* of a child *class*, which will be examined in a latter iteration of the algorithm.

   The engineer can add new supplementary *operations* if he or she deems necessary.

6. Semi-define a new HOORA *transaction* for the target class, for every incoming or outgoing Σ-signal to the target system, only if the signal originates from or truly terminates at the target system. This means that this step is not taken for signals which apparently originate from a plug-out or terminate at a plug-in.

   Name the *transaction* according to the Σ-signal. Each *transaction* name must be unique. At this point the description of the *transaction* will define either an initiating *class* or a receiving *class*.

[24] Thus, previous output of the algorithm, which concerns the parent system, must be amended.

7. Define a HOORA *aggregation relationship* between the target class and another *class* whose corresponding Σ-system includes the target system.

8. Define *association* and/or *specialization relationships* between the target class and other *classes*, where and if the engineer concludes that it is appropriate.

9. If the target system is the last Σ-system in this level of abstraction, then consider the subsequently lower level of abstraction and go to step 3. If this is not true, then stay in the same level and go to step 3. If there are no more Σ-systems to be examined, in any abstraction level, then go to the next step.

10. Complete the definitions of the *transactions* in the produced OO model so that each *transaction* has an originating and a terminating *class*. This can be achieved by searching for pairs of semi-defined *transactions* from previous steps, bearing the same name, and then producing a single *transaction* definition for each pair.

    There must also not be any *operations* belonging to any *classes* without corresponding *transactions* that trigger them. If there were any supplementary *operations* defined, the engineer, at this point, should define corresponding *transactions* for them.

11. Terminate.

When the algorithm terminates the engineer should be able to produce complete *class relationship* and *class interaction diagrams (CRDs and CIDs)* as described in [8].

Transactions, which appear in *CIDs*, are *part of* the dynamic model's domain or $dmodel_{p1}$ as defined in the pseudocode in section 4.1. Thus we claim that algorithm$_\alpha$ defines the static OO model and, in addition, it *partially* defines the dynamic OO model.

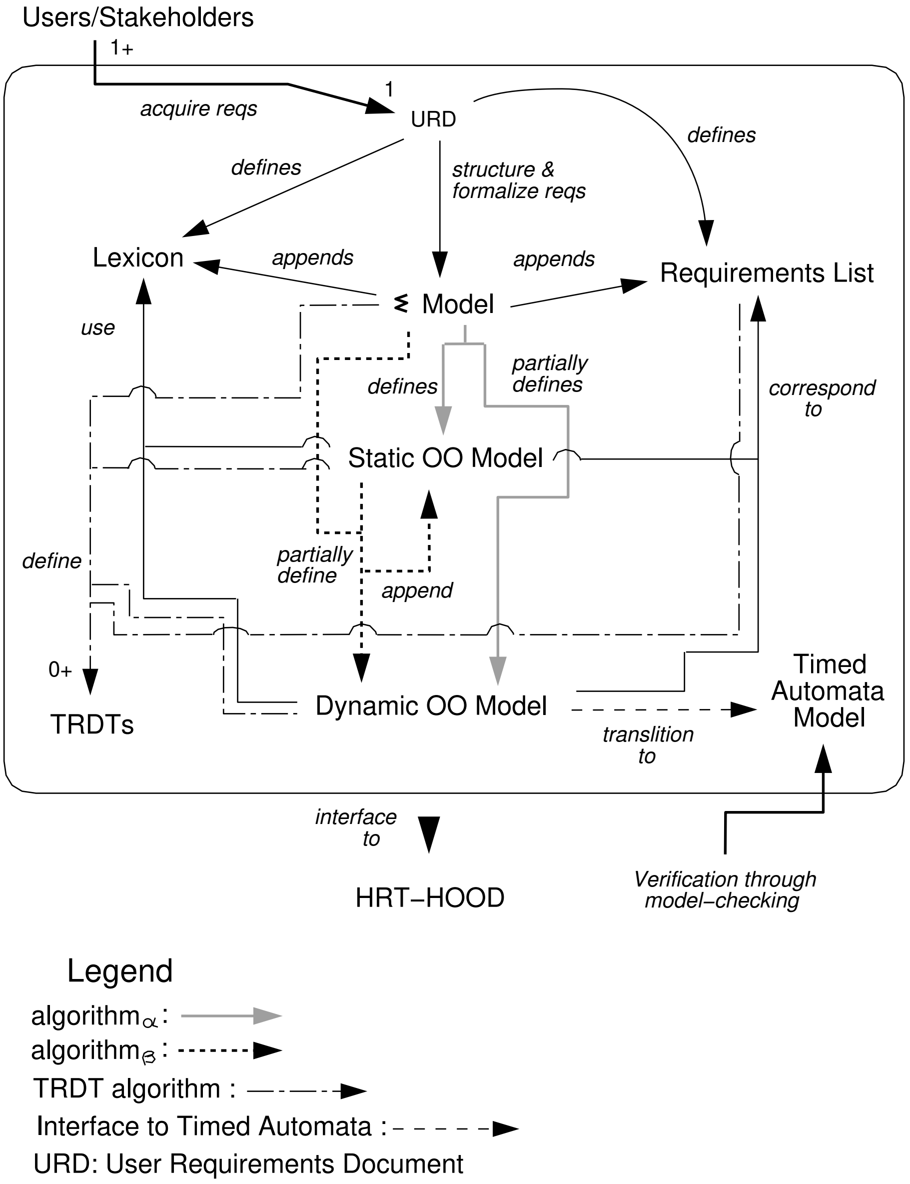

Figure 5: Algorithm$_\alpha$ and Algorithm$_\beta$ in the context of HRT-HOORA.

The role of algorithm$_\alpha$ within the context of HRT-HOORA is highlighted in figure 5.

4.3 Algorithm$_\beta$

1. Identify for which *non-external classes* found in the static OO model, *state transition diagrams (STDs)* will be constructed. Possible candidates are the most abstract classes and classes with important real-time behaviour.

2. Acquire a class from those identified in step 1, which has not been already examined.

   For the purpose of this algorithm we will refer to this class as the **target class**, its corresponding Σ-sytem as the **target system** and the under construction STD the **target STD**.

3. Identify the *state variables* of the target class, which in most cases will already be attributes of the same class. If this is not the case, append the static OO model with the *state variables* as attributes of the target class. The state variables should be normalized, that is, semantically independent of each other.

4. Identify the possible acceptable *states* for the target class by analyzing the *state variables*. Each *state* will represent a different set of values for the *state variables*. Not all combinations of values will be significant or contextually related to our view of the system, and therefore can be ignored. Insert the resulting states into the target STD, adding a state named 'IDLE', to be used as an initial state for the STD. Later this might be matched with another state and thus be merged into it.

5. Each separate Logic statement found in the target system will be considered, at least initially, as a *state transition*. If the engineer deems appropriate, a trivial or out-of-context statement will not become a state transition.

6. Try to initiate each *transition* from each possible state. Depending on the semantics of the *state* and of the *transition*, this, in most cases, will not be acceptable.

7. Try to terminate each initiated *transition* from the previous step, at each possible state, including its initiating state. Depending on the semantics of the *state* and of the *transition*, this, in most cases, will not be acceptable.

   If there is a state to be matched with the 'IDLE' state, this should be done at this point.

8. At this point each completed transition will have a corresponding Logic statement found in the target system. For the purpose of this algorithm we will refer to this statement as the **target statement**. In most cases *a)*the transition will be triggered solely by an action in the target statement which, as we know from Σ notation, *corresponds to a signal*. This signal, in turn, will correspond to a transaction already specified in the static object-oriented model. In other cases *b)*a condition will be found in conjuction with the action or *c)*a condition will be the sole cause for the transition.

   In cases *(b)* and *(c)* the condition becomes a *guard condition* for the transition in question. The *guard condition* will be shown as a small circle at the beginning of the transition arrow. In case *(c)* the transition itself can be named in plain English or not at all. In cases *(a)* and *(b)* the transition can be named in plain English but preferably the following naming template should be used:

   S(*SenderClass*, *ReceiverClass*): *transaction name*

9. Each *state* can have multiple *entry actions*, depending each time on which transition is responsible for the state change. For every individual transition these *entry actions* will be identical to the primitive actions triggered in the target statement. An *entry action* will be shown

as a small rectangle at the end of the corresponding transition arrow.

10. There will be *transitions* with special timing constraints that should be attached to them. These constraints are found in the 'Constraints' part of the target system. For such *transitions* the naming template will be extended as follows:

    S(*SenderClass*, *ReceiverClass*): *transaction name* {*timing constraint reference number*}

11. In order to prevent data congestion in the *STD*, *guard conditions*, *entry actions* and timing constraints could be recorded in a corresponding index table [25]. Each index table shall have only one corresponding *STD*.

12. If there are more classes to be acquired from those identified in step 1 then go to step 2.

13. Terminate.

When the algorithm terminates the engineer should be able to produce complete *state transition diagrams (STDs)* as described in [8] with minor modifications, and their corresponding index tables. The application of algorithm$_\beta$ (following previous application of algorithm$_\alpha$) to a Σ model, will conclude the transformation to a complete object-oriented model.

The role of algorithm$_\beta$ within the context of HRT-HOORA is graphicaly depicted in figure 5.

[25] The format of an STD index table will be accurately shown by means of subsequent case studies.

5 Case Study: The Generalized Railroad Crossing problem

5.1 The definition

In this section a Σ Notation model will be developed for a certain benchmark problem. This problem is called the Generalized Railroad Crossing (GRC) [13, 14]. For the sake of completeness the definition of the problem is quoted in the following text.

> The system to be developed operates a gate at a railroad crossing. The railroad crossing I lies in a region of interest R, i.e., $I \subseteq R$. A set of trains travel through R on multiple tracks in both directions. A sensor system determines when each train enters and exits region R. To describe the system formally, we define a gate function $g(t) \in [0, 90]$, where $g(t) = 0$ means the gate is down and $g(t) = 90$ means the gate is up. We define a set $\{\lambda_i\}$ of *occupancy intervals*, where each occupancy interval is a time interval, during which, one or more trains are in I. The *ith* occupancy interval is represented as $\lambda_i = [\tau_i, \nu_i]$, where τ_i is the time of the *ith* entry of a train into the crossing when no other train is in the crossing and ν_i is the first time since τ_i that no train is in the crossing (i.e., the train that entered at τ_i has exited as have any trains that entered the crossing after τ_i).
>
> Given two constants ξ_1 and ξ_2, $\xi_1 > 0$, $\xi_2 > 0$, the problem is to develop a system to operate the crossing gate that satisfies the following two properties:
>
> **Safety Property:** $t \in \cup_i \lambda_i \Rightarrow g(t) = 0$ (The gate is down during all occupancy intervals.)
> **Utility Property:** $t \notin \cup_i [\tau_i - \xi_1, \nu_i + \xi_2] \Rightarrow g(t) = 90$ (The gate

is up when no train is in the crossing.)

5.2 The Σ notation model

The model in this section specifies a solution to the GRC problem. Before that, we must note the following:

- The gate itself is a hardware component which can accept signals from its respective software controller to execute the actions *open* or *close*. When an action is completed the appropriate signal (GateUp or GateDown) will be sent as feedback to the software controller.
- The sensors are also hardware and have two functions; to sense a train entering into region R and to sense a train exiting from crossing I.
- In both the gate and sensor Σ specifications, the functionality is modelled as recommended hardware operations. If these functions are not implemented as recommended, the implementation of the entire model will most probably exhibit erroneous behaviour.
- The following constants, which affect ξ_1 and ξ_2, should be introduced to the model, according to Heitmeyer and Lynch [13]:
  - reach_{max}, which is an upper bound on the time from when a train enters R until it reaches I. Respectively, reach_{min} is the lower bound.
  - t_{up}, which is an upper bound on the time to raise the gate completely. Respectively, t_{down} is an upper bound to lower the gate completely. These times are likely to be the same.

Also, the following model comes with the requirements traceability information. Note that the text of the original user requirements is not written in the model but it can be found in table 4, so that the reader has a more concise view of these requirements. These kind of tables can be used freely by the engineer, when the model gets too big or too complicated for the

Key	Requirement
(0,0)	The system to be developed operates a gate at a railroad crossing
(0,1)	The railroad crossing I lies in a region of interest R, i.e.,$I \subseteq R$
(0,2)	A set of trains travel through R on multiple tracks in both directions
(0,3)	A sensor system determines when each train enters region R and exits crossing I
(0,4)	$t \in \cup_i \lambda_i \Rightarrow g(t) = 0$ The gate is down during all occupancy intervals
(0,5)	$t \notin \cup_i [\tau_i - \xi_1, \nu_i + \xi_2] \Rightarrow g(t) = 90$ The gate is up when no train is in the crossing

Table 4: Original requirements

reader to view the requirements efficiently. When the Σ notation is used in conjuction with HRT-HOORA, then the role of these tables is carried out by the *requirements list* [8]. A depiction of the requirements derivation network for the following Σ model can be found in figure 7.

The interaction described in the following model is visualized in figure 6.

[ENV$_A$] Σ: RegionC [SensorC, GateC]

(1,0,[(0,0),(0,1),(0,2),(0,3),(0,4),(0,5)]):/There will be a Region Controller which will include a Sensor Controller and a Gate Controller/

Local Variables

(3,0,[(2,0),(2,1),(2,2),(2,12)]):/The RegionC keeps track of how many trains are in region R/

TrainsInside : $\aleph$

reach$_{max}$, reach$_{min}$: TIME

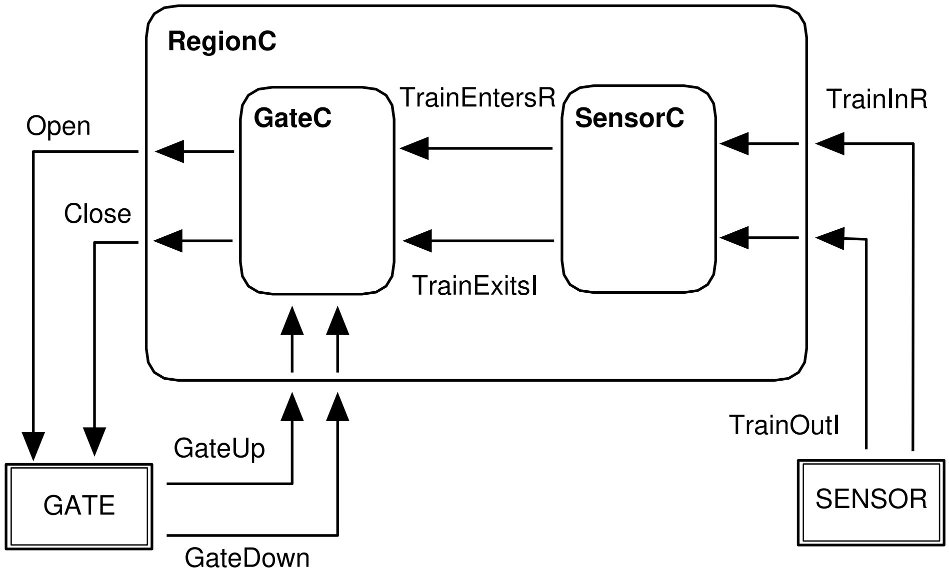

Figure 6: Interaction as described in the model.

Interaction

(3,1,[(2,0),(2,5)]):/The RegionC will be informed when a train enters region R/
Sensor(S: [TrainInR]) ▷ RegionC(plug-in: [TrainInR])

(3,2,[(2,1),(2,2),(2,6)]):/The RegionC will be informed when a train exits crossing I/
Sensor(S: [TrainOutI]) ▷ RegionC(plug-in: [TrainOutI])

(3,3,[(2,2),(2,7)]):/The RegionC can instruct the Gate to open/
RegionC(plug-out: [Open]) ▷ Gate

(3,4,[(2,0),(2,8)]):/The RegionC can instruct the Gate to close/
RegionC(plug-out: [Close]) ▷ Gate

(3,5,[(2,3),(2,9)]):/The RegionC will be informed when the Gate is fully open/

Gate(S: [GateUp]) $\triangleright$ RegionC(plug-in: [GateUp])

(3,6,[(2,4),(2,10)]):/The RegionC will be informed when the Gate is fully closed/
Gate(S: [GateDown]) $\triangleright$ RegionC(plug-in: [GateDown])

Logic

(2,0,(1,0)):/When a train enters region R, the RegionC will update the number of trains and subsequently instruct the Gate to close/
[TrainInR] $\rightharpoonup$ [Increase-by-one(TrainsInside)];[(S: [Close])]

(2,1,(1,0)):/When a train exits crossing I, the RegionC will update the number of trains/
[TrainOutI] $\rightharpoonup$ [Decrease-by-one(TrainsInside)]

(2,2,(1,0)):/If when a train exits crossing I, there are no more trains left inside, then the RegionC will instruct the Gate to open/
$\Big(\; \forall i, j : @(\downarrow[\text{Decrease-by-one(TrainsInside)}], i) =$
$@((\text{TrainsInside}= 0), j) \; \Big) \rightharpoonup [(\text{S: [Open]})]$

(2,3,(1,0)):/The state of the GateC will be updated when the Gate is fully open/
[GateUp] $\rightharpoonup$ [set(GateC.gatestate, up)]

(2,4,(1,0)):/The state of the GateC will be updated when the Gate is fully closed/
[GateDown] $\rightharpoonup$ [set(GateC.gatestate, down)]

Constraints

/The following are constraints on ξ_1 and ξ_2/

$\text{reach}_{min} < \text{reach}_{max}$

$\text{reach}_{min} > \text{GateC.t}_{down}$

$\xi_2 \geq \text{GateC.t}_{up}$

$\xi_1 \geq \text{GateC.t}_{down} + \text{reach}_{max} - \text{reach}_{min}$

$\forall i, j :$

(2,11,(1,0)):/The Gate must be fully closed before ξ_1 time passes after a train enters region R/

$@(\uparrow[\text{TrainInR}], i) < @((\text{S: [GateDown]}), i) \leq @(\uparrow[\text{TrainInR}], i) + \xi_1$

$\wedge$

(2,12,(1,0)):/For the Gate to be fully open, at least ξ_2 time should pass after the last train left crossing I/

$@((\text{S: [GateUp]}), j) \geq @((\text{TrainsInside} = 0), j) + \xi_2$

[RegionC] Σ: SensorC [ ] *(1,0)*

Interaction

(4,1,(3,7)):/The SensorC will be informed by the RegionC when a train enters region R/

RegionC(plug-in: [TrainInR]) $\triangleright$ SensorC

(4,2,(3,8)):/The SensorC will be informed by the RegionC when a train exits crossing I/

RegionC(plug-in: [TrainOutI]) $\triangleright$ SensorC

(4,3,[(3,7),(3,9),(3,10)]):/The GateC will be informed by the SensorC when a train enters region R/

SensorC(S: [TrainEntersR]) $\triangleright$ GateC

(4,4,[(3,8),(3,11)]):/The GateC will be informed by the SensorC when a train exits crossing I/
SensorC(S: [TrainExitsI]) $\triangleright$ GateC

Logic

(3,7,(2,0)):/When a train enters region R, the SensorC will update the number of trains and subsequently inform the GateC of the arrival/
[TrainInR] $\rightharpoonup$ [Increase-by-one(RegionC.TrainsInside)];[(S: [TrainEntersR])]

(3,8,(2,1)):/When a train exits crossing I, the SensorC will update the number of trains and subsequently inform the GateC of the departure/
[TrainOutI] $\rightharpoonup$ [Decrease-by-one(RegionC.TrainsInside)];[(S: [TrainExitsI])]

Constraints

(3,14,[(2,11),(2,12)]):/Keep the proper temporal sequence in SensorC/
$\forall i, j$:
@(↑[TrainInR], i) < @(↑[Increase-by-one(RegionC.TrainsInside)], i) < @(↓[Increase-by-one(RegionC.TrainsInside)], i) < @(↑[TrainEntersR], i)
$\wedge$
@(↑[TrainOutI], j) < @(↑[Decrease-by-one(RegionC.TrainsInside)], j) < @(↓[Decrease-by-one(RegionC.TrainsInside)], j) < @(↑[TrainExitsI], j)

[RegionC] Σ: GateC [ ] *(1,0)*

Local Variables

t_{up}, t_{down} : TIME

(4,0,[(3,9),(3,10),(3,11),(3,12),(3,13)]):/The GateC keeps track of the Gate's state/
gatestate : {up, down ,going-up, going-down}

Interaction

(4,3)
SensorC(S: [TrainEntersR]) ▷ GateC

(4,4)
SensorC(S: [TrainExitsI]) ▷ GateC

(4,5,(3,11)):/The GateC can instruct the RegionC to open the Gate/
GateC(S: [Open]) ▷ RegionC(plug-out: [Open])

(4,6,[(3,9),(3,10)]):/The GateC can instruct the RegionC to close the Gate/
GateC(S: [Close]) ▷ RegionC(plug-out: [Close])

(4,7,(3,12)):/The GateC will be informed by the RegionC when the Gate is fully open/
RegionC(plug-in: [GateUp]) ▷ GateC

(4,8,(3,13)):/The GateC will be informed by the RegionC when the Gate is fully closed/
RegionC(plug-in: [GateDown]) ▷ GateC

Logic

(3,9,(2,0)):/If a train enters region R and the Gate is either fully open or opening then the GateC will instruct the RegionC to close the Gate and the state of GateC will be updated/
$\big($ [TrainEntersR] $\wedge$ $\big($ (gatestate = 'up') $\vee$ (gatestate = 'going-up')$\big)$ $\big)$ $\rightharpoonup$ [(S: [Close])] $\|$ [set(gatestate, going-down)]

(3,10,(2,0)):/If a train enters region R and the Gate is either fully closed or closing then the GateC will instruct the RegionC to close the Gate/
$\big($ [TrainEntersR] $\wedge$ $\big($ (gatestate = 'down') $\vee$ (gatestate = 'going-down')$\big)$ $\big)$ $\rightharpoonup$ [(S: [Close])]

(3,11,(2,2)):/If when a train exits crossing I, there are no more trains left inside, then the GateC will instruct the RegionC to open the Gate and the state of GateC will be updated/
$\big(\forall i, j : @(\uparrow[\text{TrainExitsI}], i) = @((\text{RegionC.TrainsInside}= 0), j) \big)$ $\rightharpoonup$ [(S: [Open])] $\|$ [set(gatestate, going-up)]

(3,12,(2,3))
[GateUp] $\rightharpoonup$ [set(gatestate, up)]

(3,13,(2,4))
[GateDown] $\rightharpoonup$ [set(gatestate, down)]

(3,14,(2,2)):/If when a train exits crossing I, there are one or more trains left inside, then the GateC will take no action/
$\big($ [TrainExitsI] $\wedge$ (RegionC.TrainsInside $\geq$ 1) $\big)$ $\rightharpoonup$ *no action*

Constraints

(3,15,[(2,11),(2,12)):/Keep the proper temporal sequence in GateC/
$\forall i, j$:
$@(\uparrow[\text{TrainEntersR}], i) \leq @(\uparrow[\text{Close}], i) \leq @(\uparrow[\text{GateDown}], i)$
$\wedge$
$@((\text{RegionC.TrainsInside}= 0), j) \leq @(\uparrow[\text{Open}], j) \leq @(\uparrow[\text{GateUp}], j)$

[ENV$_A$] Σ: Sensor [ ]

(1,1,[(0,0),(0,1),(0,2),(0,3)]):/A sensor network interacts with the Region controller/

Interaction

(3,1)
Sensor(S: [TrainInR]) ▷ RegionC(plug-in: [TrainInR])

(3,2)
Sensor(S: [TrainOutI]) ▷ RegionC(plug-in: [TrainOutI])

Logic

(2,5,(1,1)):/A sensor will detect incoming trains to the region, and inform the region controller appropriately (hardware function)/
Train enters in Region R $\rightharpoonup$ [(S: [TrainInR])]

(2,6,(1,1)):/A sensor will detect outgoing trains from the crossing, and inform the region controller appropriately (hardware function)/
Train exits from crossing I $\rightharpoonup$ [(S: [TrainOutI])]

[**ENV**$_A$] Σ: **Gate** [ ] *(0,0)*

Interaction

(3,3)
RegionC(plug-out: [Open]) $\triangleright$ Gate

(3,4)
RegionC(plug-out: [Close]) $\triangleright$ Gate

(3,5)
Gate(S: [GateUp]) $\triangleright$ RegionC(plug-in: [GateUp])

(3,6)
Gate(S: [GateDown]) $\triangleright$ RegionC(plug-in: [GateDown])

Logic

(2,7,(1,2)):/When a signal to open the gate is received, the gate will act appropriately (hardware function)/
[Open] $\rightharpoonup$ Stop present operation, if any, and raise gate

(2,8,(1,2)):/When a signal to close the gate is received, the gate will act appropriately (hardware function)/
[Close] $\rightharpoonup$ Stop present operation, if any, and lower gate and if already closed, ignore

(2,9,(1,2)):/When the gate is fully raised, the region controller will be informed appropriately (hardware function)/
Gate fully raised $\rightharpoonup$ [(S: [GateUp])]

(2,10,(1,2)):/When the gate is fully lowered, the region controller will be informed appropriately (hardware function)/
Gate fully lowered $\rightharpoonup$ [(S: [GateDown])]

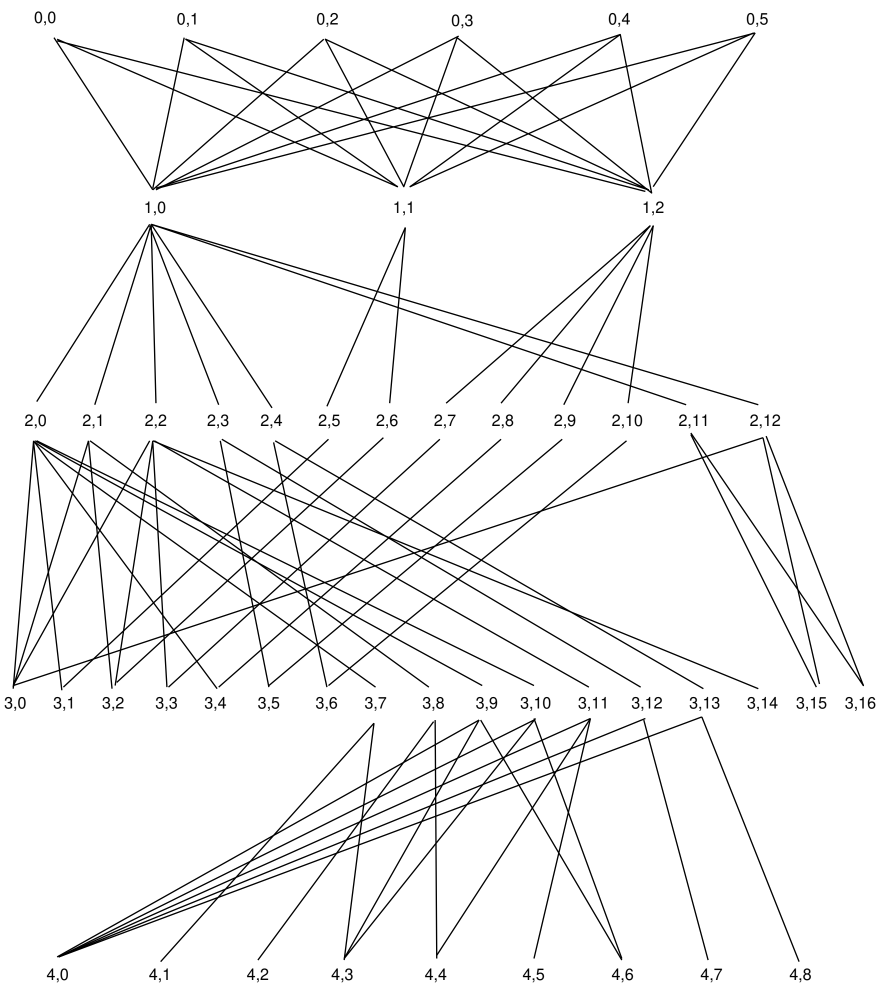

Figure 7: Derivation of requirements for the GRC problem.

5.3 Verification of consistent timing

Following the completion of the Σ model, we have to verify its timing coherence, before moving on to subsequent object-oriented analysis. In order to achieve that we will use the properties of the extended Σ notation, discussed in section 3.3.1 and attempt to apply the rule for primary timing consistency in a Σ model, as described in section 3.3.2.

After close inspection of the Σ model, we come easily to the conclusion that our main concern is the *RegionC* Σ-system and its timing properties.

To that effect, we should be able to verify the existence of primary timing coherency between *RegionC*, *SensorC* and *GateC*.

Therefore, according to the rule of timing consistency among parent and children Σ-systems, we have to show that the following expression is always true:

$$deset(\sigma_{RegionC}(\vartheta_4)) \Rightarrow deset(\sigma_{SensorC}(\vartheta_4)) \wedge deset(\sigma_{GateC}(\vartheta_4))$$

As described in section 3.3.2, we can attempt to prove the following expression instead:

$$\neg deset(\sigma_{SensorC}(\vartheta_4)) \vee \neg deset(\sigma_{GateC}(\vartheta_4)) \Rightarrow \neg deset(\sigma_{RegionC}(\vartheta_4))$$

After elaboration on the antecedent part, we reach the following extensive expression, comprising of four disjuncted parts:

$$\begin{gathered}
@(\uparrow [\text{TrainInR}], i) \geq \\
@(\uparrow [\text{Increase-by-one(RegionC.TrainsInside)}], i) \geq \qquad (1) \\
@(\downarrow [\text{Increase-by-one(RegionC.TrainsInside)}], i) \geq \\
@(\uparrow [\text{TrainEntersR}], i) \\
\vee \\
@(\uparrow [\text{TrainOutI}], j) \geq \\
@(\uparrow [\text{Decrease-by-one(RegionC.TrainsInside)}], j) \geq \qquad (2) \\
@(\downarrow [\text{Decrease-by-one(RegionC.TrainsInside)}], j) \geq \\
@(\uparrow [\text{TrainExitsI}], j) \\
\vee \\
@(\uparrow [\text{TrainEntersR}], i) > @(\uparrow [\text{Close}], i) \geq \qquad (3) \\
@(\uparrow [\text{GateDown}], i) \\
\vee \\
@((\text{RegionC.TrainsInside} = 0), j) > @(\uparrow [\text{Open}], j) \geq \qquad (4) \\
@(\uparrow [\text{GateUp}], j)
\end{gathered}$$

By first inspection we can see that propositions 1 and 2 are always false, since, by definition, the beginning of an action can not occur after its conclusion. Bare in mind though that our aim is to prove mathematically the existence of timing coherency between the parent and its children Σ-systems.

So, from propositions (1) and (3) we can derive that:

$$@(\uparrow[\text{TrainInR}], i) \geq @(\uparrow[\text{GateDown}], i) \tag{5}$$

We also know *by definition* (section 3.1.4) that:

$$@(\uparrow[\text{GateDown}], i) = @((\text{S: }[\text{GateDown}]), i) \tag{6}$$

From propositions (5) and (6), we clearly reach the conclusion that:

$$@(\uparrow[\text{TrainInR}], i) \geq @((\text{S: }[\text{GateDown}]), i) \tag{7}$$

Proposition (7) is though, in direct conflict with the first conjunct of the *RegionC* constraints, which states that:

$$@(\uparrow[\text{TrainInR}], i) < @((\text{S: }[\text{GateDown}]), i) \leq @(\uparrow[\text{TrainInR}], i) + \xi_1$$

By examining propositions (2) and (4) in a similar fashion, we can reach a proposition with a conflicting truth value to the second conjunct of the *RegionC* constraints.

Nevertheless, since the constraints of *RegionC* are in conjunction with each other, it would suffice to show the negation of one of the two conjuncts, in order to imply the negation of the whole. Thus, we have successfully reached our initial goal of proving primary timing coherency between *RegionC* and its children Σ-systems.

5.4 Building the OO model

5.4.1 Applying algorithm$_\alpha$

We will now follow the step-by-step application of algorithm$_\alpha$ to the Σ model for the Generalized Railroad Crossing problem. The numbers denote the execution of the corresponding algorithmic step.

Iteration 1

1 → Special Σ-system **ENV**$_A$ becomes **Class Environment**

2 → We consider the level in which **Σ: Sensor, Σ: Gate** and **Σ: RegionC** are present

3 → **Σ: Sensor** becomes **Class Sensor**

4 → Define new attribute (state variable):

- *status:*{*TrainEnteredRegion, TrainExitedCrossing*}

5 → No action

6 → Define transactions TrainInR and TrainOutI initiating from **Class Sensor**

7 → Define aggregation relationship between **Class Environment** and **Class Sensor** (**Class Environment** *has a* **Class Sensor**)

8 → No action

9 → Go to step 3

Iteration 2

3 → **Σ: Gate** becomes **Class Gate**

4 → Define new attribute (state variable):

- *status:*{*GateUp, GateDown, RaisingGate, LoweringGate*}

5 → Define operations Open and Close belonging to **Class Gate**

6 → Define transactions Open and Close being received at **Class Gate** and transactions GateUp and GateDown intiating from **Class Gate**

7 → Define aggregation relationship between **Class Environment** and **Class Gate** (**Class Environment** *has a* **Class Gate**)

8 → No action

9 → Go to step 3

Iteration 3

3→ Σ**: RegionC** becomes **Class RegionC**

4→ Local variables *TrainsInside*, $reach_{max}$ and $reach_{min}$ become attributes of **Class RegionC**, with the same types and semantics

5→ Define operations TrainInR, TrainOutI, GateUp and GateDown belonging to **Class RegionC**. These operations are covered by one or more child classes of **Class RegionC**

6→ No action

7→ Define aggregation relationship between **Class Environment** and **Class RegionC** (**Class Environment** *has a* **Class RegionC**)

8→ No action

9→ Consider subsequently lower level of abstraction and go to step 3

Iteration 4

3→ Σ**: SensorC** becomes **Class SensorC**

4→ No action

5→ Define operations TrainInR and TrainOutI belonging to **Class SensorC**. These operations cover the corresponding operations belonging to **Class RegionC**

6→ Define transactions TrainInR and TrainOutI being received at **Class SensorC** and transactions TrainEntersR and TrainExitsI initiating from **Class SensorC**

7→ Define aggregation relationship between **Class RegionC** and **Class SensorC** (**Class RegionC** *has a* **Class SensorC**)

8→ Define association relationship between **Class SensorC** and **Class Sensor** (**Class SensorC** *controls* **Class Sensor**)

9→ Go to step 3

Iteration 5

3→ Σ**: GateC** becomes **Class GateC**

4→ Local variables t_{up}, t_{down} and *gatestate* become attributes of **Class GateC**, with the same types and semantics

5$\rightarrow$ Define operations TrainEntersR and TrainExitsI belonging to **Class GateC**. Also define operations GateUp and GateDown belonging to **Class GateC**, which cover the corresponding operations belonging to Class RegionC
6$\rightarrow$ Define transactions TrainEntersR, TrainExitsI, GateUp and GateDown being received at **Class GateC** and transactions Open and Close initiating from **Class GateC**
7$\rightarrow$ Define aggregation relationship between **Class RegionC** and **Class GateC** (**Class RegionC** *has a* **Class GateC**)
8$\rightarrow$ Define association relationship between **Class GateC** and **Class Gate** (**Class GateC** *controls* **Class Gate**)
9$\rightarrow$ Go to next step
10$\rightarrow$ Define transactions TrainInR and TrainOutI originating from **Class Sensor** and terminating at **Class SensorC**.
Define transactions TrainEntersR and TrainExitsI originating from **Class SensorC** and terminating at **Class GateC**.
Define transactions Open and Close originating from **Class GateC** and terminating at **Class Gate**.
Define transactions GateUp and GateDown originating from **Class Gate** and terminating at **Class GateC**.
11$\rightarrow$ Terminate.

Using the results from the application of algorithm$_\alpha$ to the GRC problem we can construct CRDs and CIDs for our OO model. See figures 8, 9, 10, 11, 12.

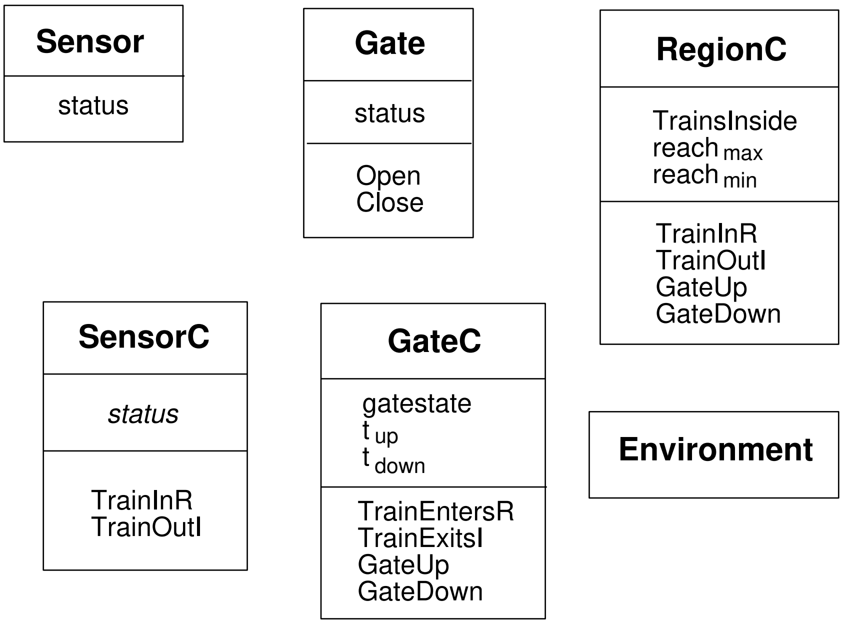

Figure 8: Produced classes in detail.

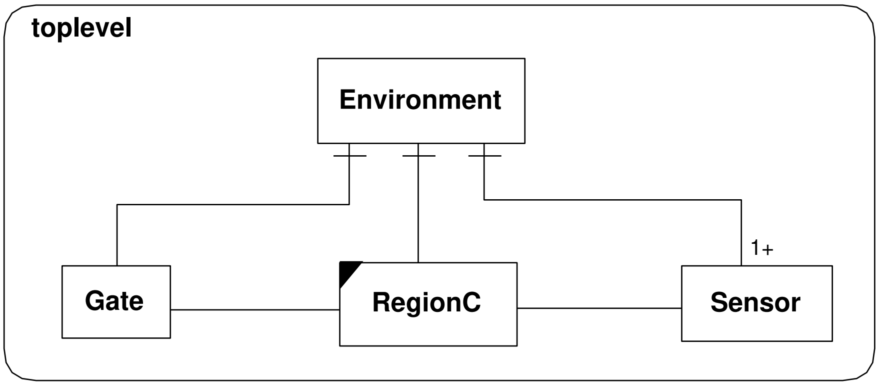

Figure 9: Toplevel CRD.

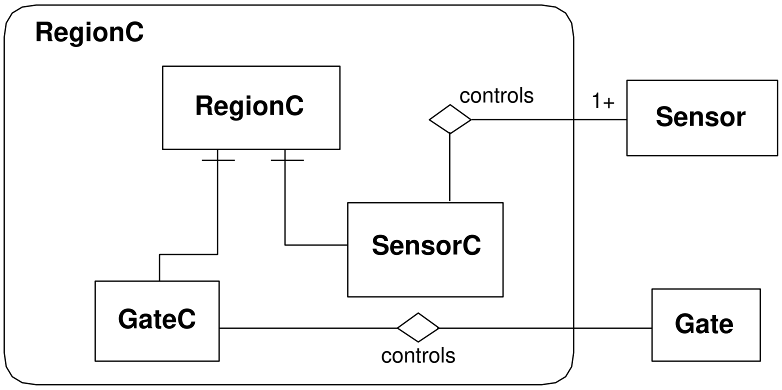

Figure 10: CRD for RegionC.

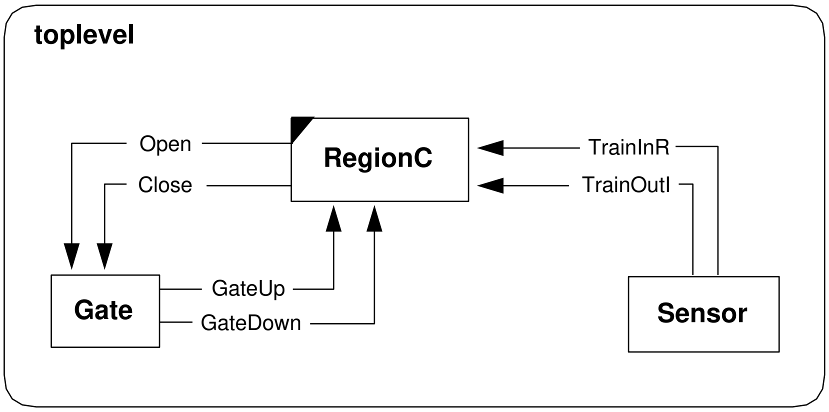

Figure 11: Toplevel CID.

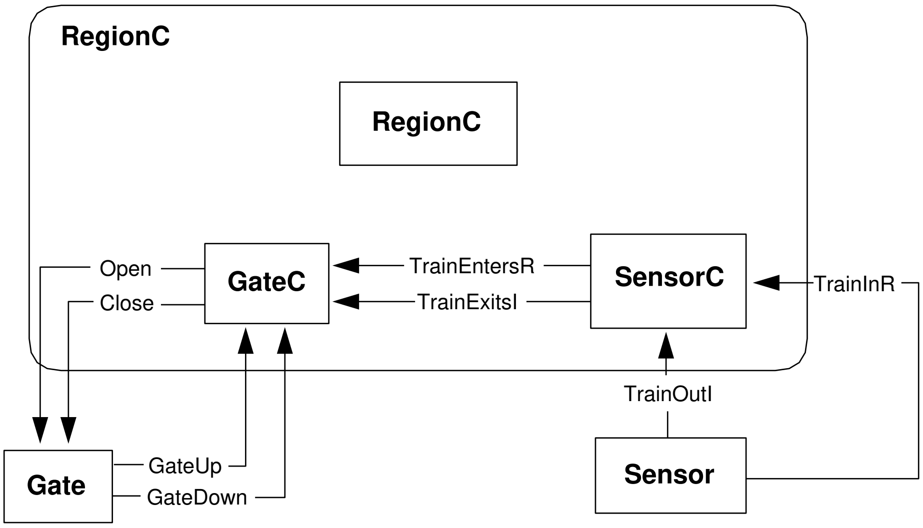

Figure 12: CID for RegionC.

5.4.2 Applying algorithm$_\beta$

We will now follow the step-by-step application of algorithm$_\beta$ to the Σ model for the Generalized Railroad Crossing problem and the static OO model produced by algorithm$_\alpha$. The numbers denote the execution of the corresponding algorithmic step.

Iteration 1

1 $\rightarrow$ The classes identified are the **Class RegionC** and **Class GateC**.
2 $\rightarrow$ Acquire **Class RegionC**.
3 $\rightarrow$ The state variables are:

- TrainsInside: $\aleph$

4 $\rightarrow$ Significant and acceptable states for **Class RegionC** are the following:

- TrainsInside > 0: One or more trains are inside the region. (state 1)
- TrainsInside $= 0$: No trains are inside the crossing. (state 2)
- State 'IDLE'. (by default)

5 $\rightarrow$ All five Logic statements found in the target system will become transitions, described here in plain English:

- Transition when a train enters the region. (transition 1)
- Transition when a train exits the crossing. (transition 2)
- Transition when the last train exits the crossing. (transition 3)
- Transition when the gate is fully closed. (transition 4)
- Transition when the gate is fully open. (transition 5)

The particular labelling of states (previous step) and transitions will be used only for the referencing purposes of this step-by-step procedure and will not apply outside of that.

6→ Match state (2) to state 'IDLE'. State (2) is now the initial state. Transition (1) can be initiated from state (2) and state (1). Transition (2) can be initiated from state (1). Transition (3) can be initiated from state (1). Transition (4) can be initiated from state (1). Transition (5) can be initiated from state (2).

7→ Transitions (1) can terminate at state (1). Transition (2) can terminate at state (1). Transition (3) can terminate at state (2). Transition (4) can terminate at state (1). Transition (5) can terminate at state (2).

8→ Only transition (3) will have a guard condition. Name the transitions according to the rules (naming template). The transition names can be seen in the STD (figure 13).

9→ Attach entry actions to states for each transition. See STD for results (figure 13).

10→ Transitions (4) and (5) will have timing constraints attached to them. Update the transition names according to the extended template.

11→ Label accordingly the guard conditions, entry actions and timing constraints and subsequently build the index table for the RegionC STD (table 5).

12→ Go to step 2

Iteration 2

2→ Acquire **Class GateC**.

3→ The state variables are:

- gatestate: {up, down ,going-up, going-down}

4→ The states are already perceived in a descrete way by the state variable. They are:

- Gate is fully up. (state 1)

- Gate is going up. (state 2)
- Gate is going down. (state 3)
- Gate is fully down. (state 4)
- State 'IDLE'. (by default)

5→ All six Logic statements found in the target system will become transitions, described here in plain English:

- Transition when a train enters the region and the Gate is either open or opening. (transition 1)
- Transition when a train enters the region and the Gate is either closed or closing. (transition 2)
- Transition when the last train exits the crossing. (transition 3)
- Transition when a train exits but more trains left inside. (transition 4)
- Transition when the gate is fully open. (transition 5)
- Transition when the gate is fully closed. (transition 6)

The particular labelling of states (previous step) and transitions will be used only for the referencing purposes of this step-by-step procedure and will not apply outside of that.

6→ Transitions (1) can be initiated from state (1) and state (2). Transitions (2) can be initiated from state (3) and state (4). Transitions (3) can be initiated from state (3) and state (4). Transition (4) can be initiated from state (3) and state (4). Transition (5) can be initiated from state (2). Transition (6) can be initiated from state (3).

7→ Transitions (1) can terminate at state (3). Transition (2) initiated from state (3) can terminate at state (3). Transition (2) initiated from state (4) can terminate at state (4). Transitions (3) can terminate at state (2). Transition (4) initiated from state (3) can terminate at state (3). Transition

(4) initiated from state (4) can terminate at state (4). Transition (5) can terminate at state (1). Transition (6) can terminate at state (4).
Set automatic transition from state 'IDLE' to state (1), thus matching the two states.
8→ Name the transitions according to the rules (naming template).
Transitions (1)(2) and (3) will have guard conditions. The transition names can be seen in the STD (figure 14).
9→ Attach entry actions to states for each transition. See STD for results (figure 14).
10→ The temporal ordering constraint is caprured by the STD's structure.
11→ Label accordingly the guard conditions and entry actions and build the index table for the GateC STD (table 6).
12→ Go to next step.
13→ Terminate.

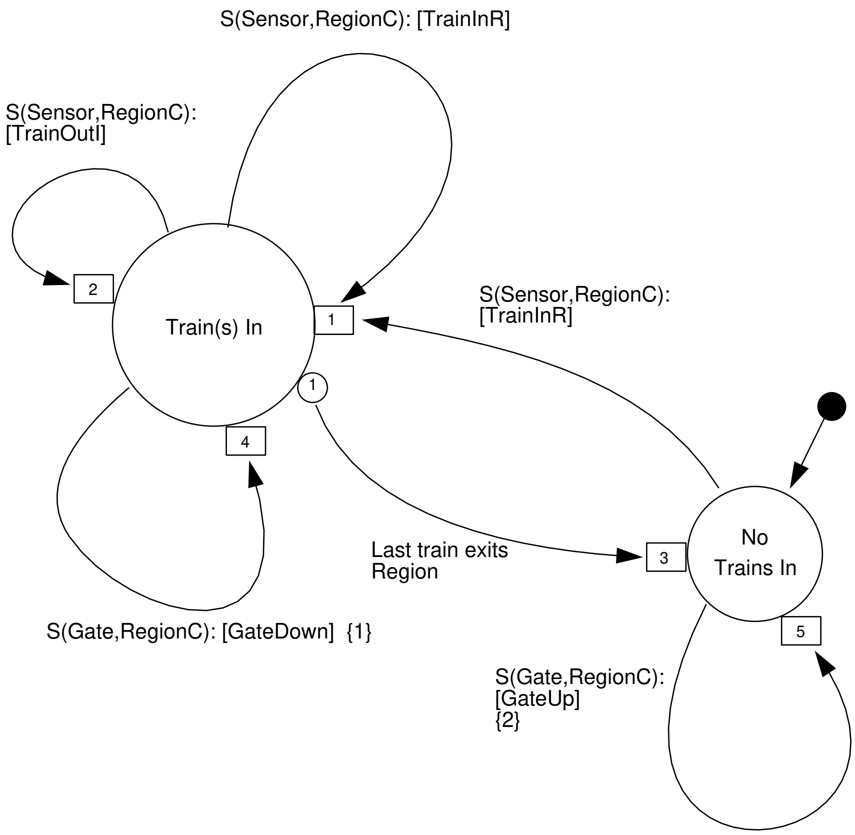

Figure 13: RegionC STD.

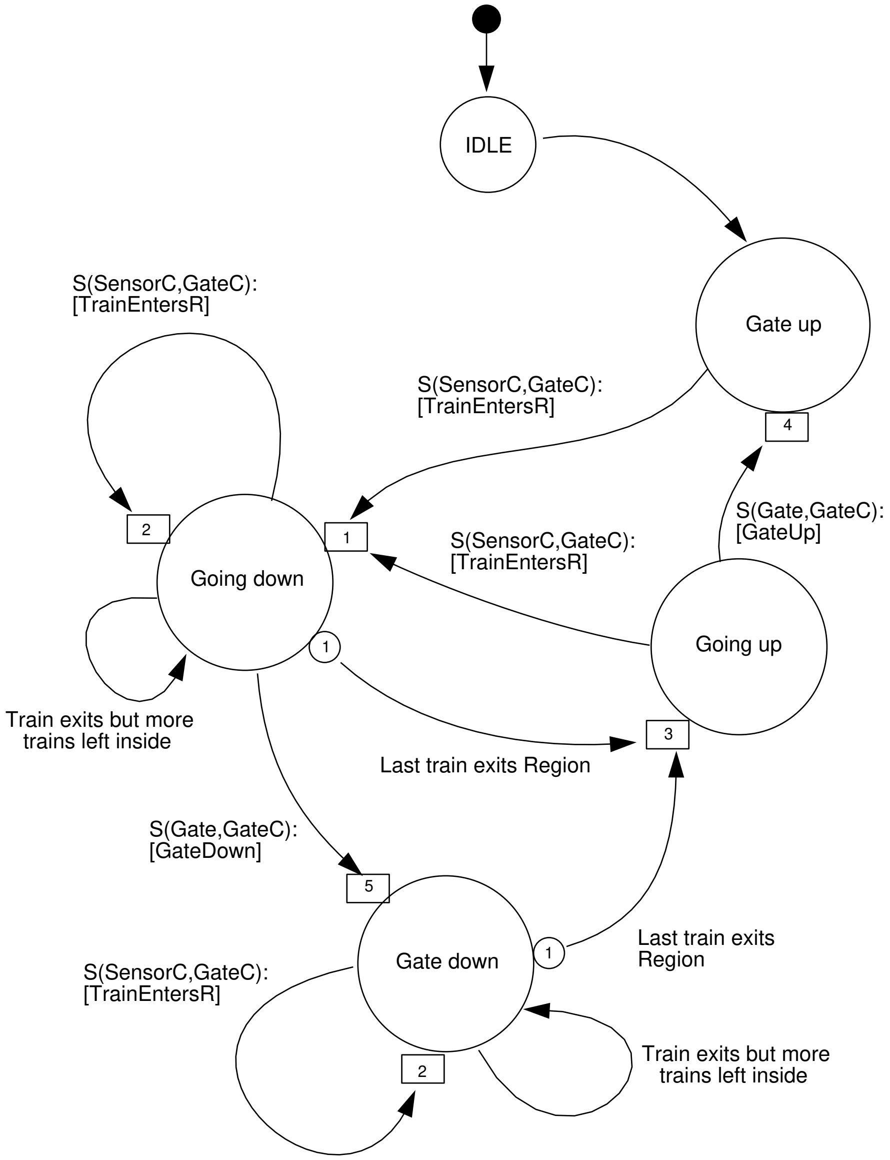

Figure 14: GateC STD.

Index table for RegionC STD	
Entry Actions	
1	[Increase-by-one(TrainsInside)];[(S:[Close])]
2	[Decrease-by-one(TrainsInside)]
3	[(S:[Open])]
4	[set(GateC.gatestate, down)]
5	[set(GateC.gatestate, up)]
Guard Conditions	
1	$\forall i, j$: @($\downarrow$[Decrease-by-one(TrainsInside)], i) = @((TrainsInside= 0), j)
Constraints	
1	@((S: [TrainInR]), i) < @((S: [GateDown]), i) $\leq$ @((S: [TrainInR]), i) + ξ_1
2	@((S: [GateUp]), j) $\geq$ @((TrainsInside = 0), j) + ξ_2

Table 5: Index table for RegionC STD

Index table for GateC STD	
Entry Actions	
1	[(S: [Close])] $\parallel$ [set(gatestate, going-down)]
2	[(S: [Close])]
3	[(S: [Open])] $\parallel$ [set(gatestate, going-up)]
4	[set(gatestate, up)]
5	[set(gatestate, down)]
Guard Conditions	
1	gatestate $\in$ {up, going-up}) $<$ @((S: [TrainEntersR]), i), @((S: [TrainEntersR]), i)]
2	gatestate $\in$ {down, going-down}) $<$@((S: [TrainEntersR]), i), @((S: [TrainEntersR]), i) $>$
3	$\forall i, j$: @($\uparrow$[TrainExitsI], i) = @((RegionC.TrainsInside= 0), j)

Table 6: Index table for GateC STD

6 Conclusions

The purpose of the Σ Notation is to be an integral part of the HRT-HOORA method or to stand alone as a complete precursor to object-oriented static and dynamic analysis. The main function of the notation, is to provide the engineer with a coherent, rigorously defined, way of formalizing and structuring informally expressed user requirements.

Modelling user requirements, mainly for embedded hard real-time systems, using this notation, is based on the construction of different Σ-systems. These formal entities capture in a controled way different requirements of the entire system and cooperate through explicit signaling.

They are related to each other by aggregation, thus creating a hierarchy of Σ-systems, which defines aspects of the entire system through different levels of abstraction. The engineer also has the ability to fashion simple aggregation into pure decomposition, by introducing extra complementary Σ-systems, in order to prevent aggregated entities and their components from functioning at the same level of abstraction. Decomposition will be employed mandatorily further down the project lifecycle, as HRT-HOOD takes over the design process.

Aggregation relationships, combined with the real-time and mathematical underpinning of the notation, can assist the expression, decomposition and analysis of timing constraints, ultimately resulting into an early consistency verification and validation of the expected real-time behaviour.

Validation of various properties captured in Σ models is also assisted by the expression in natural language (as in the Albert and Albert II languages) of all the derived requirements, with one-to-one coupling to their formal counterparts. That enhances the user-to-engineer communication, helping especially the engineer to get the all-important feedback from the user. Moreover, each requirement can be refined using a certain derivation protocol and is also coupled with a traceability enhancement stamp that can be constructed under rigorous guidelines. This enables the engineer to explicitly identify individual, derived or not, requirements and to keep a precise trace of their derivation histories.

Finally, major future development plans for the Σ Notation include support for *a)*systems operating in distributed environments, and for *b)*different operational modes.

References

[1] A. Burns and A. Wellings. *HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems.* Elsevier Science B.V., 1995.

[2] Alan M. Davis. *Software Requirements: Objects, Functions and States.* Prentice-Hall International, Inc., Englewood Cliffs, N.J., 1993.

[3] Bruce P. Douglass. *Real-time UML: Developing efficient objects for embedded systems.* Object Technology. Addison Wesley Longman, Inc., March 1998. ISBN 0-201-32579-9.

[4] E. Dubois, P. Du Bois, M. Petit, and S. Wu. ALBERT: A Formal Agent-Oriented Requirements Language for Distributed Composite Systems. In *Proceedings of the workshop on Formal Methods for Information System Dynamics (part of CAiSE 94)*, June 1994.

[5] E. Dubois, P. Du Bois, and J.M. Zeippen. A Formal Requirements Engineering Method for Real-Time, Concurrent, and Distributed Systems. In *Proceedings of the ICSE Workshop on Formal Methods Application in Software Engineering Practice*, April 24-25, 1995.

[6] Eugène Dürr, Stephen Goldsack, and Jan van Katwijk. Real Time Formal Specification using VDM++. In Max Mühlhäuser, editor, *Special Issues in Object-Oriented Programming*, pages 434–442. dpunct - Verlag für digitale Technologie, 1996. Proceedings of the ECOOP96 Workshop.

[7] E2S Software Engineering. *HOORA: Method User Manual*, October 1993. Document reference: E2S/OORA/WP1/UM. ESA contract number: 10068/92/NL/JG(SC).
http://www.estec.esa.nl/wmwww/WME/oot/index.html.

[8] E2S Software Engineering. *HOORA: Reference Manual*, October 1993. Document reference: E2S/OORA/WP1/REF. ESA contract number: 10068/92/NL/JG(SC).
http://www.estec.esa.nl/wmwww/WME/oot/index.html.

[9] J. Galle. HOORA: A Method for Hierarchical Object Oriented Requirements Analysis - Final Report. Technical report, E2S Software Engineering, March 1995. Document reference: E2S/OORA/WP5/FINREP. ESA contract number: 10068/92/NL/JG. http://www.estec.esa.nl/wmwww/WME/oot/index.html.

[10] J. Galle. HOORA: Hierarchical object-oriented requirements analysis for the European Space Agency. *Journal of Object Oriented Programming*, 9(3), June 1996.

[11] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[12] D. Harel et al. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16(4):403–413, April 1990.

[13] C. Heitmeyer and N. Lynch. The Generalized Railroad Crossing: A Case Study in Formal Verification of Real-Time Systems. In *Proceedings of the 15th IEEE Real-Time Systems Symposium*. IEEE Computer Society Press, December 1994.

[14] C. Heitmeyer and N. Lynch. The Generalized Railroad Crossing: A Case Study in Formal Verification of Real-Time Systems. Technical report, Lab. for Comp.Sci., MIT, Cambridge, MA, 1994. Technical Report MIT-LCS-TM-511.

[15] F. Jahanian and A. Ka-Lau Mok. Safety Analysis of Timing Properties in Real-Time Systems. *IEEE Transactions on Software Engineering*, SE-12(9), September 1986.

[16] F. Jahanian and A. Ka-Lau Mok. Modechart: A Specification Language for Real-Time Systems. *Transactions on Software Engineering*, 20(12):933–947, December 1994.

[17] Bernard Jungen. *ALBERT II: Reference Manual*, March 1996. Version 1.2, part of the CAT project.

[18] J. Moffett, J. Hall, A. Coombes, and J. McDermid. A Model for a Causal Logic for Requirements Engineering. *Journal of Requirements Engineering*, 1(1):27–46, March 1996.

[19] Rational Software Corporation. *UML Notation Guide*, January 1997. Version 1.0. Most recent updates in: http://www.rational.com/.

[20] Rational Software Corporation. *UML Process - Specific Extensions*, January 1997. Version 1.0. Most recent updates in: http://www.rational.com/.

[21] Rational Software Corporation. *UML Semantics*, January 1997. Version 1.0. Most recent updates in: http://www.rational.com/.

[22] Rational Software Corporation. *UML Semantics Appendix M1 - UML Glossary*, January 1997. Version 1.0. Most recent updates in: http://www.rational.com/.

[23] Rational Software Corporation. *UML Summary*, January 1997. Version 1.0. Most recent updates in: http://www.rational.com/.