Modelling and simulation of a telecommunication application using Constraint Programming

Noric Couderc
Abstract

Modelling and simulation of a telecommunication application using Constraint Programming

Noric Couderc

The complexity of telecommunication applications is constantly increasing, therefore, specialized tools are needed to test the real-world implementation of such applications. Finding test cases can be time consuming. This paper presents an approach in test cases generation using constraint programming mainly focused on time requirements between messages. A model is presented for requirements of the ETWS (Earthquake and Tsunami warning system).

An abstraction of the first model is also presented to ease modelling of similar telecommunication applications. The abstract model is found to scale nicely, making it possible to generate communication logs for many notifications. A summary of the drawbacks of the abstract model is also presented.
Contents

1 Introduction and background information 7
  1.1 Introduction ........................................... 7
  1.2 Constraint Programming ................................. 9
    1.2.1 Decision Variables ................................. 9
    1.2.2 Propagation ....................................... 10
    1.2.3 Branching ......................................... 10

2 The 4G-LTE standard and ETWS 13
  2.0.4 Protocols and Architectures ......................... 13
  2.0.5 Devices and architecture components ................. 14
  2.0.6 Types of messages .................................. 15
  2.1 The Warning Message delivery procedure ............... 15
  2.2 Packet types ......................................... 16
    2.2.1 The Paging Message ............................... 17
    2.2.2 SystemInformationBlockType1 .................... 18
    2.2.3 SystemInformationBlockType10 ................... 18
    2.2.4 SystemInformationBlockType11 ................... 18
  2.3 Scheduling ........................................... 19
  2.4 Mandatory and optional packets ....................... 20

3 Modelling correct behaviour 23
  3.1 The Model ............................................ 23
    3.1.1 Instance Data and Derived Constraints ........... 23
    3.1.2 Decision Variables ................................ 24
    3.1.3 Constraints for each notification ................. 25
    3.1.4 Constraints for each type of message .............. 28
    3.1.5 Branching Heuristics ............................. 29
4 Abstraction Layer

4.1 Recurring Patterns .................................................. 31
4.2 Diverging patterns .................................................... 32
4.3 General architecture: an abstraction ................................. 32
    4.3.1 Introducing repetitions ....................................... 34
4.4 Pipe-lining models .................................................... 34
4.5 Replacement ........................................................... 35
    4.5.1 Instance Data and Derived Constraints ....................... 36
    4.5.2 Decision Variables ........................................... 37
    4.5.3 Constraints .................................................. 37
    4.5.4 Branching Heuristics ......................................... 39
    4.5.5 Output filtering ............................................. 39
4.6 Abstract message generation .......................................... 39
    4.6.1 Messages .................................................. 40
    4.6.2 Instance Data and Derived Constraints ....................... 40
    4.6.3 Decision Variables ........................................... 41
    4.6.4 Constraints .................................................. 42
    4.6.5 Branching Heuristics ......................................... 43
    4.6.6 Updating intervals ........................................... 44
4.7 Modelling ETWS again ................................................ 44
4.8 Further than timestamps .............................................. 46
    4.8.1 SIB1 additional data ......................................... 46
    4.8.2 SIB11 additional data ......................................... 47

5 Conclusions .................................................................. 51

Bibliography .................................................................. 54
Chapter 1

Introduction and background information

1.1 Introduction

Networks have always been a part of mankind. The invention of language itself reflected the will of humans to communicate and build relationships. As spoken languages spread around the globe, humans have been able to use social behaviour and coordinate their actions, in order to survive. A strategy that, as we know now, was successful. As language became the norm for sharing information within humans, language quickly expanded, reaching outstanding levels of complexity. Language has been the origin of most of what mankind is today composed of. Science, religion, and politics are all products of the ability that humans have to share ideas via artificially constructed communication systems.

Mankind however needed a long time — until the XIX century — to gain the ability to have conversations with people being far away from the speaker. However, the invention of the telephone was the starting point of a new way of communicating. Making it easy for humans to communicate naturally, no matter the distance, became crucial.

Since the Second World War, removing wires from the communication process has been one of the main goals of telecommunication engineers. This evolution of telecommunications led to what humans use nowadays, networks of mobile phones connected via radio base stations, communicating via radio waves with their peers and, more generally, the Internet. As any communication process, these mobile phones need a common language to speak with each other. More generally, the whole set of interconnected networks needs to follow a set of rules that makes communication possible. The last standard
describing the architecture and protocols used within the network of mobile devices — which will be called User Equipments in this paper — is called the LTE 4G Standard.

The problem engineers meet today is that communication between electronic devices is not easily understandable by human beings. This communication has reached an impressive level of complexity, and so did the network architecture. After building machines able to communicate, humans now have to build tools to be able to understand how they communicate, and ease deployment and extension of a network that is not that simple to understand fully any more. As an example, it is sometimes necessary to simulate correct and incorrect behaviour in the communication between the UE and the entry point of the network a radio base station (basically a radio antenna). This simulation of correct and incorrect communication makes it possible to build automated tools that will therefore be able to make the difference between correct and incorrect behaviour in the real world.

This report (written in collaboration with Ericsson) will try to propose a solution for testing of a very small part of the 4G-LTE Standard, called Earthquake and Tsunami Warning System (ETWS). This system is used to broadcast messages in order to warn the population and fasten evacuation, in case of an incoming disaster.

Hence, the goal of this project was to provide a tool which makes it easy for engineers to simulate correct and incorrect behaviour in radio-communications between the UE and the Radio base station it is connected with. In order to do so, Constraint Programming, a technology further introduced in next section, is used.

More explicitly, an UE generates a log after reception of ETWS messages. A test harness analyses those logs in order to check for errors in communication (e.g. wrong timestamps, etc.). A previous project [1] consisted of using Constraint Programming to generate valid logs and non-valid logs in order to check if the test harness react to errors injected in the logs.

The previous project has been re-written from SICStus Prolog to C++ using the Gecode framework. (Ericsson engineers do not use Prolog). However, this version clearly lacks modularity and efficiency.

The current version to be used at Ericsson is a chunk of 2366 lines of C++ code stored in a single file. As a result, modifying the underlying model to create new test cases is a difficult task. The current model is difficult to adapt to other 4G services, and deriving test cases is also very complicated. Engineers at Ericsson wanted an abstraction layer that could be easily manipulated, modified, and used in various contexts that are different from ETWS.

The goal of the project is to provide a framework or an abstraction layer
that simplify modelling of a telecommunication application log, and make this model more flexible.

1.2 Constraint Programming

Constraint Programming is a programming paradigm, in which the user does not provide a sequence of steps to solve a problem, but instead declares constraints that apply to user-defined variables. It is mainly used to find solutions to Constraint Satisfaction Problems (CSP) and Constraint Optimisation Problems (COP). This technology is nowadays used extensively in the industry to solve such classes of combinatorial problems. This section attempts to provide a short introduction to the field of Constraint Programming, only the basic principles will be covered.

Constraint Programming has been used in dedicated programming languages, such as Prolog, but has also been implemented as libraries in order to be used in imperative or functional languages. In this chapter, Gecode, a C++ framework for Constraint Programming will be used.

1.2.1 Decision Variables

The first step of CP modelling is the definition of a number of variables. These variables are not variables as meant by programmers — i.e., names for storing values — but mathematical variables. In that sense, decision variables are defined with their domain, domain in which variables can take any value. In this context, while a variable, in the imperative programming world, is defined as follows:

\[ x = 0 \]

Decision variables are defined as:

\[ x \in \mathbb{N} \]

Of course, the statement above is a lie. Constraint programming, at least as implemented in the real world, cannot deal with such an infinite set. Therefore, the domain of decision variables is always bounded (e.g. \( 0 < x < 100 \)). However, one has to remember that decision variables reflect the mathematical meaning of the word “variable”, in the sense that it does not have a specific value at every moment.

In Constraint Programming, a solution is considered found when all decision variables have a domain containing a single value. Which therefore leads to the following question: how to get to \( x = 56 \) from \( 0 < x < 100 \)?

Propagators are the answer to that question.
1.2.2 Propagation

To shrink the domain of a decision variable enough that it only contains one value, one has to define constraints relative to the problem that one wants to solve. A propagator is the concrete application of a constraint to a decision variable. A propagator removes values not satisfying the stated constraint from the domain of concerned decision variables. This process itself is called Propagation. Sometimes this propagator leads to pruning all values from the variable domain. In this case the propagator, which has to report success or failure of propagation, reports a failure.

A constraint can be related to only one decision variable, or many decision variables. Constraints that deal with many decision variables are called global constraints, those are probably the most powerful feature of Constraint Programming. As an example, the distinct constraint apply on an array of decision variables and constraints its elements to be all pairwise different.

However, simple propagation might not be enough to reach a single value for the domain of all decision variables — e.g., stating that all decision variables have pairwise different values does not say which those values are — therefore, after a first propagation phase a choice must be made. Taking decisions is called branching.

1.2.3 Branching

Branching — also called systematic search — is the last step of the process of finding a solution. In case the first phase of propagation did not lead to a satisfactory solution, an arbitrary decision is made. This decision is in the form of a new constraint being posted, for example $x = 5$. This leads to a new context where propagators that have been posted may be able to prune more values from the domains. Hence, all propagators are ran once again, in this new context. If a propagator reports failure after its execution, that means that the choice made previously was a bad choice. Hence, backtracking must be performed, and another choice has to be made, regarding the variable. Choosing which variable to branch upon and which values should be tried for it first is a decision of the user. “Which variable to choose first” is called a variable selection heuristic and “Which value of the domain choose for the selected variable” is called a value selection heuristic. Those heuristics should not remove any solution to the model. No matter which heuristic are chosen, the total number of solutions that the model can output should stay the same. Branching choices are just a way to fasten finding of new solutions.

Constraint Programming is a general-purpose programming paradigm that can be used for any kind of constraint satisfaction problem, or con-
straint optimisation problem. In this project, Constraint Programming is used to generate a log according to constraints regarding the communication protocol used in ETWS. The following chapter will provide background information regarding telecommunication protocols, and the architecture of the E-UTRAN network, as defined in the specification of the 4G-LTE network.
Chapter 2

The 4G-LTE standard and ETWS

This Chapter is a small introduction to the architecture of the 4G-LTE standard. The architecture of the network itself is called E-UTRAN for Evolved Universal Terrestrial Radio Access Network.

The vocabulary used in telecommunication applications, unfortunately, involves many acronyms and abbreviations. In order to ease the reading of the report, this chapter also contains a small lexicon. Taking a peek at the lexicon while reading the report is strongly recommended. For more information, please refer to the specification.

2.0.4 Protocols and Architectures

Here are defined the various protocols and architectures that are relevant to the content of the paper.

**LTE:** *Long Term Evolution*, commonly referred to as 4G LTE, a standard of wireless communication for mobile phones and terminals.

**PWS:** *Public Warning System*, one of the services included in the LTE standard in order to send public warning to the users of the network.

**ETWS:** *Earthquake and Tsunami Warning System*, a subset of PWS used to send warnings regarding Earthquakes and Tsunamis to users of the LTE Network.

### 2.0.5 Devices and architecture components

The E-UTRAN architecture is composed of many components — actual physical devices — that are connected to each other.

**UE:** *User Equipment*, a device connected to the LTE Network, e.g. a mobile phone or a computer.

**eNodeB:** *Evolved Node B*, another name for a radio base station. An UE is connected to an eNodeB, communicating via radio waves. Several UEs can be connected to one eNodeB, this connection between eNodeB and UE is the only one of the whole architecture that has to be wireless.

**MME:** *Mobility Management Entity*, Another layer of the U-TRAN Architecture: eNodeBs are connected to an MME. A MME may interface with several eNodeBs.

**CBC:** *Cell Broadcast Centre*, MMEs are connected to a cell broadcast centre.

**CBE:** *Cell Broadcast Entity*, CBCs are all connected to a CBE, which can be located at the site of content provider.

The basic network structure of PWS is depicted in figure 2.1 which is extracted from the 3GPP Technical Specification [2] p. 11.

![Figure 2.1: The E-UTRAN PWS Architecture](image-url)
2.0.6 Types of messages

Sending an ETWS notification to UEs follows a procedure that involves most of the components of the network. This procedure follows a tree-like architecture, from a central root to the leaves of the tree, here the UEs.

In ETWS a notification is called WRWR, a Write Replace Writing Request.

**WRWR:** Write-Replace Warning Request, Request sent from an CBC to MMEs.

**WRWC:** Write-Replace Warning Confirm, Sent from an MME to a CBC to confirm broadcast of the request in the lower level of the network (i.e. to eNodeBs).

2.1 The Warning Message delivery procedure

Network registration and security procedures are performed previously to the Warning Message delivery procedure (i.e. the UE is already connected to a valid 4G network). Here follows a simplified version of the Warning Message delivery procedure [2, p. 21-23]. Figure 2.2 denotes the architecture.

1. CBE sends emergency information to the CBC. The CBC shall authenticate the request.

2. The CBC identifies which MMEs need to be contacted and sends a WRWR (Write-Replace Warning Request) containing the message to be broadcast to MMEs.

3. The MMEs that are contacted by the CBC send a WRWC (Write-Replace Warning Confirm) that indicates to the CBC that the MME has started distributing the warning message to eNodeBs.

4. The CBC forwards the WRWC to the CBE, confirming that the warning message is actually being distributed.

5. The MME forwards the WRWR to eNodeBs. MMEs use a Tracking Area ID List to determine which eNodeBs in the delivery area should be contacted. If this list is empty, then all eNodeBs connected to the MME are contacted.
6. An eNodeB may receive same message from multiple MMEs. In case of redundant messages, only the first one received will be broadcast. If there is a warning broadcast message already ongoing, the eNodeB shall immediately replace the existing message by the newer one.

7. The eNodeB broadcasts the message frequently according to the attributes set by the CBC that originated the warning message distribution.

8. If the UE (User Equipment) has been configured to receive warning messages, then the UE indicates the contents of the warning message to the user.

9. If the Send Warning-Message-Indication parameter was present in the WRWR, the MMEs shall forward the Broadcast Scheduled Area Lists in a Write-Replace Warning Indication(s) to the CBC.

   The Broadcast Scheduled Area Lists contains the aggregation of lists of cells where the message has been actually broadcast. This list is generated by the MME by aggregating Broadcast Completed Area Lists it receives from eNodeBs.

   A Broadcast Completed Area List is generated by an eNodeB and contains the list of cells the Warning Message has been broadcast to.

10. eNodeBs return a Write-Replace Warning Response message. From this message the MME determines the success or failure of the delivery and creates a trace record.

   Figure 2.2 extracted from the specification [2, p. 22] describes the Warning Message Delivery Procedure.

   One has to keep in mind that this project is however focused on communication between the UE and an eNodeB, the eNodeB sends many messages to the UE during broadcast of an ETWS notification. This communication is described in more detail in section 2.2.

2.2 Packet types

The communication between an UE and an eNodeB follows several rules of scheduling [3, p. 25], such as periodicity and ordering of packets. Those rules are further introduced in section 2.3. However, it is necessary to mention the different types of packets one has to deal with. System Information is divided in two sections:
Figure 2.2: The E-UTRAN PWS Procedure

- **MasterInformationBlock (MIB):** General information (Not relevant in the scope of the project).

- **SystemInformationBlocks (SIBs):** System Messages. SIBs other than SystemInformationBlockType1 are carried through SystemInformation (SI) messages.

As ETWS is a service, the communication is made through SI messages. When there is a change in the network information, the UEs have to be notified about this change. To do so, Paging Messages are used.

### 2.2.1 The Paging Message

The Paging Message is used for the notification of one or more UEs [3, p. 139]. A Paging Message including the `systemInfoModification` informs UEs that the network information will change at the next modification period boundary. No further details are provided (e.g. regarding which system information will change).
Default paging cycle. The defaultPageCycle (dPC) is a parameter which determines the time (in milliseconds) between two consecutive reads of paging messages. For example, every defaultPagingCycle the UE wakes up for 1ms, to read a paging message. In the context of the project, paging message are supposed to contain only one information: and ETWS indication stating if there are ETWS messages incoming or not.

2.2.2 SystemInformationBlockType1

systemInformationBlockType1 is not specific to ETWS and contains information relevant when evaluating if an UE is allowed to access a cell, it also defines the scheduling of other system information [3, p. 157].

systemInformationBlockType1 includes a value tag: systemInfoValueTag, that indicates if a change had occurred in the SI (System Information) messages. The SIB1 message contains an schedulingInfoList, this schedulingInfoList schedules future reception of SIB10 and SIB11. The schedulingInfoList contains the periodicities of SIB messages that will follow. It has to be mentioned that, if a secondary notification is broadcast, and a primary notification starts being broadcast in the meantime, the SIB1 message will provide periodicity about the primary notification beginning as well as the periodicity of messages of the secondary notification being broadcast.

2.2.3 SystemInformationBlockType10

systemInformationBlockType10 contains the primary notification of ETWS [3, p. 182]. It identifies the source and type of ETWS notification (i.e. Tsunami and/or Earthquake).

An SIB10 message contains:

- messageIdentifier: Identifies the source and type of ETWS notification.
- serialNumber: Identifies the variations of an ETWS notification.
- warningType: Identifies the warning type of ETWS primary notification (Tsunami and/or Earthquake).

2.2.4 SystemInformationBlockType11

systemInformationBlockType11 contains the secondary notification of ETWS [3, p. 183]. The main difference between SIB10 and SIB11 is that SIB11 comes
in segments. It identifies the alphabet, encoding and language applied variations of an ETWS notification. It identifies the source and type of the ETWS notification.

An SIB11 message contains:

- **messageIdentifier**: Identifies the source and type of an ETWS notification.

- **serialNumber**: Identifies the variations of an ETWS notification.

- **warningMessageSegmentType**: Indicates whether the included ETWS warning message is the last segment or not.

- **warningMessageSegmentNumber**: Segment number of the ETWS warning message segment contained in the SIB. Segment numbers start from zero.

- **warningMessageSegment**: Carries a segment of the Warning Message Contents IE.

- **dataCodingScheme**: Identifies the alphabet/coding and the language applied variations of an ETWS notification.

### 2.3 Scheduling

The UE continuously check the existence of new Paging Messages every $d_{PC}$ milliseconds. However, in ETWS, Paging messages are only broadcast during a certain number of $d_{PC}$. The number of default paging cycles is an eNodeB parameter. For an incoming SIB10 Message the Paging Message is broadcast during the same amount of time as the SIB10 Message (see figure 2.3).

![Figure 2.3: The scheduling of a primary notification](image-url)
For an SIB11 Message, the Paging Message is broadcast as soon as the notification message reach the eNodeB, it keeps being broadcast until the end of the next dPC. The SIB11 Message is broadcast during the whole repetition period and the process start over a certain number of times. Those two parameters are defined in the notification the eNodeB receive from the MME. (see figure 2.4).

Moreover, to declare the end of transmission a new paging message and a new SIB1 message are sent. All SIB1 messages should contain *schedulingInfoList*, if no scheduling information is given regarding a certain type of message, then no messages of this type should exist before next SIB1 message.

### 2.4 Mandatory and optional packets

A distinction has to be made between mandatory packets, that have to be received in order to validate the transmission, and optional packets that are subject to a different set of constraints.

**Primary notifications.** In case of primary notification, a paging message and a SIB1 message have to be received consecutively every dPC. After reception of SIB1 *at least one* SIB10 message has to be received before the next paging message is received. Optional packets can be scheduled after or before the mandatory packet for each dPC. Those packets are only constrained to follow periodicity of SIB10 packets.

Figure 2.5 denotes the timing structure that packets should respect when a primary notification is broadcast.
Secondary notifications. In case of secondary notification, a paging message and a SIB1 message have to be received each broadcast. After the SIB1 message, at least \( \text{numberOfSegments} \) SIB11 messages have to be received. SIB11 packets are numbered, and SIB11 packets numbers are constrained to be in sequence: A packet numbered \( n \) has to be followed by a packet numbered \( n + 1 \), while \( n < \text{numberOfSegments} \). Otherwise, the following packet has to be numbered 0.

As an example, for \( \text{numberOfSegments} = 3 \) the only sequences allowed are:

- \([0, 1, 2]\)
- \([1, 2, 0]\)
- \([2, 0, 1]\)

After reception of the sequence, optional packets can be received. Optional packets are constrained to follow a more flexible version of the sequence constrain: their timestamps have to be consistent with their numbering. This constraint model the fact that packets can be missed, but their numbering has to be correct. For example, if segment 3 is received, segment 5 can be received afterwards, but those two segments have to be separated by 2 times the periodicity of SIB11. That means that segment 3 was received, segment 4 was sent but not read by the UE, and segment 5 was received, respecting the periodicities of SIB11.

Figure 2.6 denotes the timing structure that packets should respect when a secondary notification is broadcast.
Now that the protocol has been defined, the protocol can be modelled using constraint programming, which is the subject of Chapter 3.

In Chapter 4, a more flexible model will be proposed, trying to get rid of as many explicit references to Paging, SIB1, SIB10, and SIB11 as possible.

Figure 2.6: Mandatory and optional packets for secondary notification
Chapter 3

Modelling correct behaviour

This Chapter provides the result of the first attempt to model the ETWS system through constraint programming. Even though this chapter might help understanding of the whole system, it does not reflect some choices made in the Abstraction layer (Chapter 4), nor model completely the ETWS notification system (see last paragraph of the chapter).

3.1 The Model

3.1.1 Instance Data and Derived Constraints

In order to generate the log, the data received by the eNodeB has to be modelled. An MME Notification is encoded as a struct containing several fields. The PagingPeriod parameters denotes the time while the Paging message is broadcast. This parameter does not appear in an actual MME Message since this delay is configured in eNodeB. However, it has to be mentioned in the modelling, and the difference between the period while Paging messages are broadcast, for primary and secondary notification, has to be specified as well. As a result, primaryPagingPeriod and secondaryPagingPeriod are present in the model view of an MME message. In the real world, MME Messages only contain repetition period and number of broadcasts requested. All timestamps are in milliseconds.

- timestamp: denotes the time-stamp of the MME Message.
- type: denotes the type of notification (PRIMARY, SECONDARY or PRIMARY_AND_SECONDARY)
- primaryPagingPeriod: the Paging period of the primary notification.
• \textit{secondaryPagingPeriod}: the Paging period of the secondary notification.

• \textit{brdcestCycles}: the number of Paging cycles while the primary notification is broadcast.

• \textit{numberOfBroadcastsRequested}: the number of broadcast of the secondary notification.

• \textit{numberOfSegments}: the number of segments of the secondary notification.

• \textit{repetitionPeriod}: the repetition period of each broadcast of the secondary notification.

• \textit{periodicities}: an array of integer values being the periodicities of each type of message (0 for Paging, 1 for SIB1, 2 for SIB10 and 3 for SIB11).

\section*{3.1.2 Decision Variables}

The Model should be able to handle several notifications at once, hence some fragmentation has to be performed. For each notification, the timestamps are enclosed in arrays of decision variables, one array for each type of message. The sizes of array are described in section 3.1.3.

• \textit{PagPN}: The array of timestamps for Paging messages of primary notification.

• \textit{PagSN}: The array of timestamps for Paging messages of secondary notification.

• \textit{SIBPN}: The array of timestamps for SIB1 messages of primary notification.

• \textit{SIBSN}: The array of timestamps for SIB1 messages of secondary notification.

• \textit{SIBPrim}: The array of timestamps for SIB10 messages of primary notification.

• \textit{OptSIBPrim}: The array of timestamps for optional SIB10 messages of primary notification.

• \textit{SIBS}: The array of timestamps for SIB11 messages of secondary notification.
• **OptSIBS**: The array of timestamps for SIB11 messages of secondary notification.

Primary Paging and secondary Paging packets as well as primary and secondary SIB1 have to be separated to permit simultaneous broadcast of **PRIMARY** AND **SECONDARY** notifications. The variables prefixed by *Opt* are used for the handling of optional packets.

### 3.1.3 Constraints for each notification

This section describes the constraints applied to each one of the broadcast notifications.

**Number of mandatory messages.** The lengths of arrays of variables are defined as follows

- **Number of Paging messages**: A Paging message is received every \( \text{brdcstCycles} + 1 \) Paging message for notification of end of transmission.
  \[ |\text{PagPN}| = \text{brdcstCycles} + 1 \]

- **Number of SIB1 messages**: A SIB1 is received every \( \text{brdcstCycles} + 1 \) SIB1 message for notification of end of transmission.
  \[ |\text{SIB1PN}| = \text{brdcstCycles} + 1 \]

- **Number of SIB10 messages**: A SIB10 is received every \( \text{brdcstCycles} \).
  \[ |\text{SIBPrim}| = \text{brdcstCycles} \]

For secondary notifications:

- **Number of Paging messages**: A Paging message is received every broadcast:
  \[ |\text{PagSN}| = \text{numberOfBroadcasts} + 1 \]

- **Number of SIB1 messages**: A SIB1 is received every broadcast:
  \[ |\text{SIB1SN}| = \text{numberOfBroadcasts} + 1 \]

- **Number of SIB11 messages**: A numberOfSegments SIB11 segments is received every broadcast:
  \[ |\text{SIBSec}| = \text{numberOfBroadcasts} \times \text{numberOfSegments} \]
Number of optional messages. The size of those arrays is less obvious than the size of arrays for mandatory messages. Since the model has to simulate the inherent randomness of telecommunications through a physical space, a choice has been made to give a randomly determined size to these arrays (regarding some constraints of course). This behaviour will be changed in Chapter 4.

The number of optional messages received for primary and secondary notifications is at least 1. To determine the number of optional messages that can be received some calculations must be made. First, the time span where a SIB10 message can be received must be computed:

\[
\text{PrimPeriod} = (\text{primaryPagingPeriod} - \text{periodicities}_0 - \text{periodicities}_1)
\]

Knowing, this time span, one can compute the maximum number of SIB10 message that could be received during this time span.

\[
\text{maxPrim} = \text{PrimPeriod} ÷ \text{periodicities}_2
\]

The limit of number of SIB10 messages is therefore:

\[
\text{limitPrim} = \text{maxPrim} - \text{brdcstCycles}
\]

- Number of optional SIB10 messages: At least one optional message is received, at most \(\text{limitPrim}\) optional messages are received.

\[1 \leq |\text{SIBPrim}| < \text{limitPrim}\]

As previously stated, a random value is picked in this interval. For secondary notifications the same process is followed: First compute the timespan while a SIB11 message can be broadcast.

\[
\text{SecPeriod} = \text{repetitionPeriod} \cdot \text{numberOfBroadcastsRequested}
\]

Then the limit of number of optional SIB11 messages that can be received can be calculated.

\[
\text{limitSec} = \text{SecPeriod} ÷ \text{periodicities}_3 - |\text{SIB11}|
\]

- Number of optional SIB11 messages: At least one optional message, at most \(\text{limitSec}\) optional message are received.

\[1 \leq |\text{SIBSec}| < \text{limitSec}\]

A random value is picked within this interval.
Message Scheduling Constraints. For primary notifications:

- **Paging after MME Message**: The Paging time-stamp has to be higher than the time-stamp of the MME Notification.

  \[ \min(PagPN) > timestamp \]

- **Paging time-out**: First Paging must be received before a complete dPC.

  \[ \min(PagPN) \leq timestamp + dPC \]

- **Paging before SIB1**: A SIB1 message has to follow each Paging message (also true for messages at end of transmission).

  \[
  \text{for all } i \in [0, \text{brdcstCycles}]: \ PagPN_i < SIB1PN_i
  \]

- **SIB1 Time out**: SIB1 message cannot arrive more than \( \text{periodicities}_1 \) ms after Paging.

  \[
  \text{for all } i \in [0, \text{brdcstCycles}]: \\
  (SIB1PN_i - PagPN_i) \leq \text{periodicities}_1
  \]

- **SIB1 before SIB10**: A SIB10 message has to follow each SIB1 message — *not* including last SIB1 message for end of transmission —

  \[
  \text{for all } i \in [0, \text{brdcstCycles} - 1]: \ SIB1PN_i < SIBPrim_i
  \]

- **SIB10 Time out**: SIB10 message cannot arrive more than \( \text{periodicities}_2 \) ms after SIB1

  \[
  \text{for all } i \in [0, \text{brdcstCycles} - 1]: \ (SIBPrim_i - SIB1PN_i) \leq \text{periodicities}_2
  \]

For secondary notifications, some constraints have to be expressed for all messages:

- **Paging after MME Message**: The Paging timestamp has to be higher than the timestamp of the MME Notification.

  \[ PagSN_0 > timestamp \]

- **Paging time-out**: First Paging must be received before a complete dPC.

  \[ PagSN_0 \leq timestamp + \text{periodicities}_0 \]
Some constraints have to be expressed for each one of each broadcast of the secondary notification:

- **Paging before SIB1**: A SIB1 message has to follow each Paging message for all $i$ in $[0, \text{numberOfBroadcasts}]$: $\text{PagSN}_i < \text{SIB1SN}_i$

- **Paging separation**: Paging messages are at most separated by repetition period. (within one notification)
  
  for all $i$ in $[1, \text{numberOfBroadcasts}]$:
  
  $(\text{PagSN}_i - \text{PagSN}_{i-1}) \leq rPer$

- **SIB1 Time out**: SIB1 message cannot arrive more than $\text{periodicities}_1$ms after Paging.

  for all $i$ in $[0, \text{numberOfBroadcasts}]$:
  
  $(\text{SIB1SN}_i - \text{PagSN}_i) \leq \text{periodicities}_1$

- **SIB1 before first SIB11**: A first SIB11 message has to follow each SIB1 message.

  for all $i$ in $[0, \text{numberOfBroadcasts}]$:
  
  $\text{SIB1SN}_i < \text{SIBSec}_i \cdot \text{numberOfSegments}$

- **first SIB11 time-out**: First SIB11 message cannot arrive more than $\text{periodicities}_3$ms after SIB1 message.

  for all $i$ in $[0, \text{numberOfBroadcasts}]$:
  
  $(\text{SIBSec}_i \cdot \text{numberOfSegments} - \text{SIB1SN}_i) \leq \text{periodicities}_3$

- **All segments within Repetition Period**: All segments have to be received within repetition period.

  for all $i$ in $[0, \text{numberOfBroadcasts}]$:
  
  for all $j$ in $[0, \text{numberOfSegments}]$:
  
  $\text{SIBSec}_i \cdot \text{numberOfSegments} + j \leq \text{PagSN}_i + rPer$

### 3.1.4 Constraints for each type of message

Paging messages, SIB1, SIB10, and SIB11 messages are subject to a set of common constraints. The following constraints are hence applied on each type of message $t$. The array $\text{timestamps}$ is a two-dimensional array denoting the timestamps of messages of type $t$. 

28
Periodicities. Two ways of specifying periodicity of a message have to be mentioned:

1. Absolute periodicity: The timestamp of the message has to be a multiple of its periodicity specified in periodicities.

2. Relative periodicity: The timestamp of the message has to be a multiple of its periodicity relative to the first message of its kind.

For a type of message $t$, considering the timestamps of this type of message and the periodicity $p$ for this particular type of message.

- **Absolute Periodicity:**
  
  \[
  \text{for all } timestamp \text{ in } timestamps_t : \quad timestamps \equiv 0 \mod p
  \]

- **Relative Periodicity:**
  
  \[
  \text{for all } timestamp \text{ in } timestamps_t : \quad (timestamp - timestamps_{t,0}) \equiv 0 \mod p
  \]

Symmetry-Breaking Constraints. Many orderings of timestamps for each type of message are possible. To break this symmetry, message timestamps are chronologically ordered.

- **Timestamp ordering:** All timestamps for every type of message $t$ have to be ordered:

  \[
  \text{for all } i \text{ in } [1,|timestamps[t]|]: \quad timestamps_{t,i-1} < timestamps_{t,i}
  \]

3.1.5 Branching Heuristics

The variable selection heuristic is set to selecting the first unassigned value. The value selection heuristic is set to a random value within the domain of the selected variable.
Criticism Since this model attempts to focus on each notification very specifically, it is extremely difficult to specify constraints between notifications. As a result, this model does not handle replacement. Replacement can actually be implemented with C++ and imperative programming, but since replacement is a feature of the protocol that could possibly lead to many erratic behaviours, handling it via simple imperative programming is not totally satisfactory.

Moreover, this model only deals with timestamps, and does not deal with constraints regarding additional information that could come with, for example, SIB1 or SIB11 messages. This information has to be taken into account as well.
Chapter 4

Abstraction Layer

One can see in the previous chapter that many recurring patterns can be found within the model. Most of them are focused on the constraints on timestamps of messages. This chapter aims to exploit those recurring patterns in order to ease modelling of similar behaviours.

4.1 Recurring Patterns

There are many recurring patterns in the model, most of them deal with iteration on an array (or a matrix) and those recurring patterns can therefore be easily implemented in a set of functions.

**Sequencing.** In many cases, for two types of message $t_1$ and $t_2$, each message of type $t_1$ has to be followed by another message of type $t_2$, within same broadcast. For example: As a SIB1 has to be received after Paging, SIB11 messages need to be received after SIB1. Hence, for two arrays $a$ and $b$ of same length $l$, representing timestamps of messages of type $t_1$ and $t_2$:

$$
\text{for all } i \text{ in } [0, l]: a_i < b_i
$$

**Time-outs.** There is a time-out between messages of type $t_1$ and messages of type $t_2$. As a result for two arrays $a$ and $b$ of same length $l$, and a time-out value $v$:

$$
\text{for all } i \text{ in } [0, l]: (b_i - a_i) < v
$$

**Repetition periods.** It is usual that two broadcasts of the same notification have to be separated by a delay. For example: two broadcasts of the
same secondary notification have to be separated by \textit{repetitionPeriod}. This constraint is very similar to time-out, but it sets a lower bound to the time between two messages instead of setting a higher bound.

As a result for two arrays \( a \) and \( b \) of same length \( l \), and a time span \( v \):

\[
\text{for all } i \text{ in } [1, l[: (a_i - a_{i-1}) \geq v}
\]

**Periodicities.** All messages follow a periodicity constraint. Periodicities can be relative or absolute.

Absolute periodicity is defined as:

\[
\text{for all } timestamp \text{ in } timestamps : \hspace{1cm} timestamp \equiv 0 \mod periodicity
\]

There is also a more elegant way to define relative periodicity, than the definition in Chapter 3:

\[
\text{for all } t \text{ in } a : t \equiv offset \mod periodicity
\]

Where \( 0 \leq offset \leq periodicity \)

### 4.2 Diverging patterns

**Number of messages.** As mentioned previously, the number of messages per repetition is variable. It mainly depends on the minimum number of messages of each type, the sequencing between them, possible replacements, as well as their respective periodicities.

**Message structure.** All messages carry a timestamp. However, those messages also carry additional information. SIB1 carries periodicity information about SIB10 or SIB11, depending of type of notification, and SIB11 messages carry information about their number of segments. That means that each message can carry additional information with its timestamp.

### 4.3 General architecture: an abstraction

The ETWS Protocol is hard to model: The previous model includes eight decision variable arrays and approximately twenty propagators per notification. It is important to break down the ETWS protocol in a number of smaller elements.
The main problem of the ETWS simulation is the stochasticity of telecommunications applications. The model has to simulate the fact that telecommunications are never perfect and that messages can be lost, or that there can be a delay in reading the message. In order to simplify the modelling of the ETWS protocol, the choice has been made to simulate both **broadcasting** of messages by the eNodeB and **reading** of messages by the UE.

Modelling what the eNodeB is broadcasting is fairly easy since the stochastic problem has not to be taken into account. It is therefore easy to define intervals of time when a certain type of notification can be received. Starting from that idea of modelling what the eNodeB broadcasts, elements that are broadcast must be defined, for example: notifications.

Notifications that can have several types, in this case: Primary and Secondary notifications.

Primary and Secondary notifications are, for the most part, independent. Primary notifications can only be replaced by other primary notifications, same for secondary notification. As a result, primary and secondary notifications can be processed by separated models.

**The SIB1 Problem:** Surprisingly, the ETWS protocol involves a third type of notification, named **PRIMARY_AND_SECONDARY**. This type of notification is an alias for two notifications being broadcast exactly at the same time. In case of **PRIMARY_AND_SECONDARY** notifications, the eNodeB broadcasts one Paging message followed by one SIB1 message containing information about SIB10 and SIB11 messages that will follow. This behaviour includes that, in case one wants to deal with primary and secondary notifications in separated instances of the abstract model, this is possible, but this synchronisation of SIB1 messages has to be handled as well. As previously said, the Paging messages and SIB1 messages are generated two times, one for primary notifications, and again for secondary notifications. Those packets must be constrained to have same periodicities. Moreover, the SIB1 message have a content that is relative to all notifications being broadcast while the SIB1 message is broadcast. That means that if a secondary notification is broadcast, and a primary notification starts being broadcast in the meantime, then the SIB1 messages content have to reflect the fact that a primary and a secondary notification are broadcast right now. Meaning that the SIB1 message has to contain information about all ongoing broadcasts at the time it is broadcast.
4.3.1 Introducing repetitions

It is common, also, in the previous model, that the same data is broadcast many times. The number of times data is repeated has different names depending of the types of notification. For primary notifications it is called `brdcstCycles`, for secondary notifications this parameter is called `numberOfBroadcastsRequested`.

Even though those two parameters have different names, it is obvious that they describe the same thing. Therefore, the whole protocol can be defined as a sequence of minimal components (messages) that are encapsulated in larger components — In order to deal with repetition periods and number of broadcasts for example — these components are, in this model, named `repetitions`.

Here is a definition of the notification structure: A Notification is a list of repetitions $r_i$, where given a parameter `numberOfRepetitions`, $0 < i < numberOfRepetitions$, followed by a structure stating the end of transmission, if the notification has not been replaced. The end of transmission (here called $e$), is composed of one Paging message followed by one SIB1 message. A repetition is a list of messages which size is indefinite.

- Notification: $((r_0, r_1, ..., r_i, ...), e)$
- Repetition: $(m_0, m_1, ...)$
- End Of Transmission: $(Paging, SIB1)$

Repetitions have at least two timestamps, one for the beginning of the repetition, and one for the end of the repetition. The number of repetitions per notification is given as a parameter.

Each repetition is a sequence of messages. The type of each message might be different and can carry different information. However, each message, no matter its type, has a time-stamp. The number of messages of each type can be fixed or random — for example, there is only one Paging and one SIB1 per repetition — but there are constraints on timestamps of messages: since a repetition happens in a limited time window the number of messages per repetition is bounded.

4.4 Pipe-lining models

One of the main problems of modelling the ETWS protocol is its relation with randomness. The model has to simulate lost packets. This feature unfortunately does not really fit modelling with Gecode: Gecode expects
arrays to be created with a fixed size, that cannot be modified at any point during search.

To model this, and address this issue — messages disappearing randomly — it is possible to define a parallel array of boolean variables that encode if the message existed or not.

This feature leads to an issue: *global constraints*. Those constraints have been developed in an attempt to provide a constraint that can be applied on all values in the array of decision variables. These constraints can be reified, but it is impossible to define a global constraint that will be applied only on some decision variables in an array. Moreover, using oversized arrays can lead to a drastic loss in performance of the general model. It is possible to partly resolve this problem by separating creation of packets in different stages. In this context, it is possible to “toss away” the packets that have been deleted from the time line in one way or another. Each stage will start with a cleaned data from the previous stage. This avoids using too much memory, and therefore helps scaling. The whole pipeline is defined as follows:

1. eNodeB data: A first model, starting from information relative to notifications, build a structure of repetitions, following correct replacement behaviour.

2. Paging messages: Paging messages are created from repetitions. Each repetition is followed by only one paging message. Respecting periodicities and time-outs (Here the Paging messages stating end of transmission are not considered).

3. SIB1 messages: SIB1 messages are created from Paging messages. Each Paging message is followed by only one SIB1 message regarding periodicity and time-out (Again, not considering SIB1 messages for end of transmission).

4. SIB10 messages: Each SIB1 message is followed by at least one SIB10 message before next Paging message.

5. SIB11 messages: Each SIB1 message is followed by at least *numberOfSegments* SIB11 messages, before next Paging message.

The modelling of replacement is handled by a specific model.

After generation of repetitions, a general solution to the generation of packets have been proposed. Therefore, to generate messages from a new type of packets, running again the general model, with different parameters, is enough.
Figure 4.1 describes the architecture of the pipeline. Blue boxes are instances of the abstract model described in section 4.6.

4.5 Replacement

It is possible to generate correct replacement behaviour, considering some instance data very similar to the instance data provided to the model in Chapter 3. This section introduces a constraint programming model that takes a set of notifications as an input and compute a set of repetitions, as well as a set of end of transmissions, following correct replacement behaviour. This problem can be solved using only imperative programming, however, using a constraint programming model for this keeps the model mutable.

4.5.1 Instance Data and Derived Constraints

The instance data for the generation of correctly replaced notifications is defined as follows:

- $numNotif$: number of notifications.
• timestamps: Array of timestamps of MME Message for each notification.

• numberOfRepetitions: Array of number of repetitions for each notification.

• repetitionPeriod: Array of maximum length in milliseconds of each repetition (in case it has not been replaced).

These arrays must be of same length: numNotif.

4.5.2 Decision Variables

Here is the list of decision variables for the model. It has to be mentioned that these decision variables exist for each notification.

• begin: Array of timestamps where a repetition begins.

• end: Array of timestamps of end of repetition.

• replaced: Array of boolean variables stating if the repetition has been replaced or not.

• beginEOT: An array of integer decision variable stating the beginning of end Of Transmission notification for the current notification.

• endEOT: An integer decision variable stating the end of end Of Transmission notification for the current notification.

In the model, the decision variable array for a specific notification \( n \) will noted as, for example: \( \text{begin}_n \).

4.5.3 Constraints

Size of arrays. There is one begin timestamp, one end timestamp and one replacement flag for each repetition. Therefore, for each notification, all arrays have the same size, numberOfRepetitions.

\[
\text{for all } n \text{ in } [0, \text{numNotif}[:
|\text{begin}_n| = |\text{end}_n| = |\text{replaced}_n| = \text{numberOfRepetitions}_n
\]
Notification timestamps and beginnings. The first repetition of the notification $n$ starts with the timestamp of $n$.

\[ \text{for all } n \text{ in } [0, \text{numNotif}], \]
\[ begin_{n,0} = \text{timestamps}_n \]

A repetition starts exactly when the previous one ends.

\[ \text{for all } n \text{ in } [0, \text{numNotif}], \]
\[ \text{for all } r \text{ in } [1, \text{numberOfRepetitions}_n], \]
\[ begin_{n,r} = end_{n,r-1} \]

Notification ends and replacements. A repetition length cannot exceed the repetition period for its notification.

\[ \text{for all } n \text{ in } [0, \text{numNotif}], \]
\[ \text{for all } r \text{ in } [0, \text{numberOfRepetitions}_n], \]
\[ end_{n,r} \leq begin_{n,r} + \text{repetitionPeriod}_n \]

If a repetition is replaced, its end must be the beginning of the next notification.

\[ \text{for all } n \text{ in } [0, \text{numNotif} - 1], \]
\[ \text{for all } r \text{ in } [0, \text{numberOfRepetitions}_n], \]
\[ replaced_{n,r} \Rightarrow end_{n,r} = begin_{n+1,0} \]

If a repetition is replaced, the length of the repetition is not the repetition period.

\[ \text{for all } n \text{ in } [0, \text{numNotif}], \]
\[ \text{for all } r \text{ in } [0, \text{numberOfRepetitions}_n], \]
\[ replaced_{n,r} \Leftrightarrow end_{n,r} \neq begin_{n,r} + \text{repetitionPeriod}_n \]

End of transmission. The end of transmission come after the repetition of notification information.

\[ \text{for all } n \text{ in } [0, \text{numNotif}], \]
\[ \text{for all } r \text{ in } [0, \text{numberOfRepetitions}_n], \]
\[ \text{beginEOT}_n \geq end_{n,r} \]
Symmetry breaking constraints. All timestamps are lexicographically sorted.

\[
\text{for all } n \text{ in } [0, \text{numNotif}[:
\text{for all } r \text{ in } [1, \text{numberOfRepetitions}_n[:
\begin{align*}
\text{begin}_{n,r-1} & \leq \text{begin}_{n,r} \\
\text{end}_{n,r-1} & \leq \text{end}_{n,r}
\end{align*}
\]

4.5.4 Branching Heuristics

The selected variable selection heuristic is to choose the first non-assigned variable. The selected value selection heuristic is to choose the minimum value in the domain of selected variable.

4.5.5 Output filtering

This models allows repetitions of length 0, a repetition of length = 0 is a repetition that has been completely replaced. Therefore, those repetitions are of no use in the message generation model. The replacement model must therefore provide a filtering function that prevents the repetitions of length 0 to be present in the output data that the next stage of the pipeline will use as input data.

4.6 Abstract message generation

This section attempts to address common element from the models defined above into a one-fits-all solution. In order to generate a new sequence of messages, it is only needed to run this abstract model with new parameters. The most minimal component of a message is its timestamp. All types of message share this component, sometimes in addition with some other information. Therefore, the abstract model focuses on generation of timestamps of messages.

Regarding extra data, other than bare timestamps, that might be needed, this data has to be handled and constrained by the user. This fine-tuning of the abstract model is possible thanks to the fact that constrains and decision variables can be added to a model after its initial creation — but not during search — regarding additional data in the ETWS protocol, see section 4.8.
4.6.1 Messages

The following section describes relations between repetitions and the messages within each one of them, as a short reminder of constraints between message timestamps.

Paging messages. One Paging message has to exist for each repetition. The timestamp of paging message has to be after timestamp of repetition, it however has a time-out.

SIB1 messages. One SIB1 message has to exist for each Paging. The timestamp of paging message has to be after timestamp of Paging message. It also has a time-out.

SIB10 messages. There is at least one SIB10 message per SIB1. The timestamp of SIB10 messages has to be between timestamp of SIB1 of current repetition and Paging of next repetition.

SIB11 messages. There is at least \( \text{numberOfSegments} \) SIB11 messages after each SIB1. The timestamp of SIB11 messages has to be between timestamp of SIB1 of current repetition and Paging of next repetition.

4.6.2 Instance Data and Derived Constraints

In order to create correct timestamps for each type of messages, some input data must be provided to the model.

Intervals. As mentioned previously, the repetition structure is crucial. It defines very conveniently a narrow space for the timestamps to be generated. The data structure for a repetition interval is therefore called \( \text{interval} \) and has three member variables.

- \( \text{begin} \): The lower bound of the interval.
- \( \text{end} \): The upper bound of the interval.
- \( \text{type} \): The type of repetition, either \text{REPETITION} or \text{END_OF_TRANSMISSION}.

The list of intervals define the space that is still available in the timeline for creation of messages. As it has been mentioned, there are many constraints regarding sequencing of messages. Therefore, each instance of the
abstract model should be able to provide an updated version of the *intervals* list, in order to update the free space available for message scheduling.

The replacement model provides the first instance of the *intervals* list by computing the repetition intervals and filtering intervals of length $= 0$.

Since we have to generate a number of messages that has to be random — but bounded — it is also mandatory to provide an interval of number of generated messages per repetition.

- *numRep*: The number of repetitions.
- *intervals*: A list of intervals.
- *minMessages*: A list of minimum number of messages per repetition.
- *maxMessages*: A list of maximum number of messages per repetition.
- *periodicity*: The periodicity of this particular type of message.

Obviously, the first stage of the pipeline does not take any list of Timestamps as an entry. The lists *intervals*, *minMessages*, *maxMessages* must be of same size: *numRep*.

### 4.6.3 Decision Variables

Dealing with only one type of messages at a time helps greatly the formulation of decision variables and constraints for this particular type of message.

**Global decision variables.** There has to be a variable encoding the random delay before first message of current type if received. This decision variable is named *offset*.

- *offset*: A single decision variable which encodes the random offset of the first message of current type.

**Decision variables for each repetition.** Decision variables are stored in an array of array of decision variables. Those arrays are processed and constrained using data for each repetition. The whole structure can be considered as a matrix with rows of (possibly) different lengths.

- *timestamps*: The list of timestamps of messages.
- *exist*: The list of boolean variables encoding whether the message exists or not.

The lists *exist* and *timestamps* must be of same size.
4.6.4 Constraints

Number of messages. Make the arrays of existence of messages and timestamps of messages as large as possible.

\[
\text{for all } r \text{ in } [0, \text{numRep}]:
|\text{timestamps}_r| = |\text{exist}_r| = \text{maxMessages}_r
\]

Timestamps. Timestamps of messages have to fit in the current repetition:

\[
\text{for all } r \text{ in } [0, \text{numRep}]:
\text{for all } i \text{ in } [0, |\text{timestamps}_r|]:
\text{start} < \text{timestamps}_{r,i} \leq \text{end}
\]

where \(\text{start} = \text{begin}(\text{intervals}_r)\) and \(\text{end} = \text{end}(\text{intervals}_r)\).

Time-outs. There is often a time-out for messages, for example, if a Paging message is received, and there is no SIB1 message coming within the periodicity of SIB1 after the Paging message, the communication is assumed as incorrect.

\[
\text{for all } r \text{ in } [0, \text{numRep}]:
\text{for all } i \text{ in } [0, |\text{timestamps}_r|]:
\text{timestamps}_{r,i} \leq \text{begin}(\text{intervals}_r) + \text{timeout}
\]

Number of deleted messages. One has to be sure that the number of messages meet the generation requirements. If repetition has been replaced the length of the repetition might be less than what is required to schedule \(\text{minMessages}\) messages. If it is the case however, as many messages as possible should exist in this replaced notification.

Therefore, a value \(\text{length}\) must be computed, and divided by the \(\text{periodicity}\) of the current type of message to get the lower bound of messages that can exist within the notification.

\[
\text{for all } r \text{ in } [0, \text{numRep}]:
\text{for all } i \text{ in } [0, |\text{timestamps}_r|]:
\text{timestamps}_{r,i} \leq \text{begin}(\text{intervals}_r) + \text{timeout}
\]

\[
\text{for all } r \text{ in } [0, \text{numRep}]:
\text{\(l_r = \frac{\text{end}(\text{intervals}_r) - \text{begin}(\text{intervals}_r)}{\text{periodicity}}\)}
\]

The number of existing messages in the repetition must be more than the minimum of the computed lower bound and the minimum number of
messages. In Gecode the constraint count cannot be applied to arrays of boolean decision variables. One has to use the linear constraint to get the same result [4, p.65].

\[
\text{for all } r \in [0, \text{numRep}]: \\
\text{count}(\text{exist}_r = \text{true}) \geq \min(\text{minMessages}_r, l_r)
\]

Periodicities. The offset represents the random delay that exist before reception of the first message of its kind. It has to be between 0 and periodicity.

\[0 \leq \text{offset} \leq \text{periodicity}\]

After reception of the first packet of this kind, all packets following must respect this periodicity, therefore:

\[
\text{for all } r \in [0, \text{numRep}]: \\
\text{for all } i \in [0, |\text{timestamps}_r|]: \\
\text{timestamps}_{r,i} \equiv \text{offset} \mod \text{periodicity}
\]

Symmetry breaking constraints. The array of timestamps must be sorted in lexicographic order.

\[
\text{for all } r \in [0, \text{numRep}]: \\
\text{for all } i \in [1, |\text{timestamps}_r|]: \\
\text{timestamps}_{r,i-1} \leq \text{timestamps}_{r,i}
\]

4.6.5 Branching Heuristics

The offset variable is subject to a specific brancher. The value selection heuristic for the offset variable is random value selection: A random value is picked within the domain of offset.

Regarding other decision variables — timestamps and exist for each notification — the variable selection heuristic is set to first unassigned variable.

The value selection heuristic is set to minimum value for timestamps and random value for exist, so the stochastic behaviour of reading of messages is simulated.
Algorithm 1: Filtering algorithm

4.6.6 Updating intervals

The process of filtering intervals and updating it is done via imperative programming. The process of updating intervals is done in two steps.

1. Filtering the timestamps array to remove timestamps of message that officially do not exist. This process is defined in algorithm 1.

2. Modifying those timestamps regarding certain conditions. Update of intervals is described in algorithm 2.

The filtering algorithm requires exist and timestamps to be assigned (i.e. to have a domain consisting of only one value). Moreover, the filtering process might require sorting of timestamps and removal of duplicate timestamps. This can be done via the extremely useful library algorithm which is part of the standard C++ libraries. In this case, the functions std::sort and std::unique will fit.

After filtering the timestamps, messages of the next stage can be scheduled before or after the next messages.

4.7 Modelling ETWS again

To perform the modelling of the actual model, the input parameters to the different stages of the pipeline should be specified.

The instance data numRep and intervals, are obtained via the previous step of the pipeline (numRep = |intervals|).

Only the decision variables minMessages, maxMessages, and periodicity should change.

The default periodicities for each type of message in ETWS are defined as (in ms):

44
Require: \( \text{filtertype} \in \{\text{BEFORE, AFTER}\} \)

Require: \( \text{timestamps} \) has been filtered.

1: \( \text{intervals} \leftarrow \emptyset \)
2: \textbf{try all } \( i \) \textbf{ in } \text{intervals} \textbf{ do}
3: \( \text{firstInI} \leftarrow \infty \)
4: \( \text{lastInI} \leftarrow 0 \)
5: \textbf{try all } \( t \) \textbf{ in } \text{timestamps} \textbf{ do}
6: \quad \textbf{if } \text{begin}(i) \leq t \wedge t \leq \text{firstInI} \textbf{ then}
7: \quad \quad \text{firstInI} \leftarrow t
8: \quad \textbf{if } \text{end}(i) \geq t \wedge t \geq \text{lastInI} \textbf{ then}
9: \quad \quad \text{lastInI} \leftarrow t
10: \quad \textbf{if } \text{filtertype} = \text{BEFORE} \textbf{ then}
11: \quad \quad \text{end}(i) \leftarrow \text{firstInI}
12: \quad \textbf{else}
13: \quad \quad \text{begin}(i) \leftarrow \text{lastInI}
14: \quad \textbf{if } \text{end}(i) - \text{begin}(i) > 0 \textbf{ then}
15: \quad \quad \text{intervals} \leftarrow (\text{intervals}, i)
16: \textbf{return} \text{ intervals}

\begin{algorithm}

\textbf{Algorithm 2: } \text{intervals} \text{ update algorithm}

\end{algorithm}

- \( \text{periodicity}_{\text{paging}} = 1280 \)
- \( \text{periodicity}_{\text{SIB1}} = 20 \)
- \( \text{periodicity}_{\text{SIB10}} = 80 \)
- \( \text{periodicity}_{\text{SIB11}} = 320 \)

Paging.

- \( \text{minMessages} \): 1
- \( \text{maxMessages} \): 1
- \( \text{periodicity} \): \( \text{periodicity}_{\text{paging}} \)

SIB1.

- \( \text{minMessages} \): 1
- \( \text{maxMessages} \): 1
- \( \text{periodicity} \): \( \text{periodicity}_{\text{SIB1}} \)
SIB10.

- \( \text{minMessages} \): 1
- \( \text{maxMessages} \): \( \text{repetitionPeriod} \div \text{periodicity}_{\text{SIB10}} \)
- \( \text{periodicity} \): \( \text{periodicity}_{\text{SIB10}} \)

SIB11.

- \( \text{minMessages} \): \( \text{numberOfSegments} \)
- \( \text{maxMessages} \): \( \text{repetitionPeriod} \div \text{periodicity}_{\text{SIB1}} \)
- \( \text{periodicity} \): \( \text{periodicity}_{\text{SIB11}} \)

End of transmission  The handling of the end of transmissions is supported via running the abstract model again with parameters for generation of paging messages and SIB1 messages. The intervals input data has to be obtained from the repetition model, exclusively for end of transmissions.

4.8 Further than timestamps

So far the model has only dealt with timestamps of messages, because timestamp is the only component that all types of messages share. As specified previously, different types of messages can carry various data. To make sure that this additional data can be handled painlessly by the abstraction model, one solution is to handle addition of properties to the model itself, on the fly.

A possible solution is to maintain a traditional C++ map (see `std::map`), Mapping from a string (property name) to a decision variable: \( \text{String} \mapsto \text{DecisionVariable} \). Hence, the \text{print} function of the model can deal with printing decision variables that has been added as properties after creation of the model. Gecode also handles declaring constraints and branchers outside of the constructor of the model.

4.8.1 SIB1 additional data

After timestamps generation, the model has provided an array of timestamps of SIB1 messages for each type of notification. It is possible to push all SIB1 messages for primary notification and all SIB1 message for secondary into one single array.
Require: \(SIB1\)Timestamps, \(SIB10\)Timestamps, \(SIB11\)Timestamps

\begin{algorithm}
1: \(SIB1Info \leftarrow (0, 0, ...)
2: \text{for } i = 0 \text{ to } i = |SIB1\)Timestamps| − 1 \text{ do}
3: \quad \begin{align*}
4: & t \leftarrow SIB1\)Timestamps\(_i \\
5: & u \leftarrow SIB1\)Timestamps\(_{i+1}
6: \text{try all } SIB10 \text{ in } SIB10\)Timestamps \text{ do}
7: \quad \begin{align*}
8: & \text{if } t \leq SIB10 \land SIB10 \leq u \text{ then}
9: & \quad SIB1Info\(_i \leftarrow SIB1Info\(_i + 1
10: \text{try all } SIB11 \text{ in } SIB11\)Timestamps \text{ do}
11: \quad \begin{align*}
12: & \text{if } t \leq SIB11 \land SIB11 \leq u \text{ then}
13: & \quad SIB1Info\(_i \leftarrow SIB1Info\(_i + 2
14: \end{algorithm}

Algorithm 3: Naive algorithm for additional info of SIB1

Unfortunately, no elegant Constraint Programming solution have been found to address this problem. It is however possible to solve the problem of SIB1 additional data by scanning the timestamps of SIB1, SIB10, and SIB11 messages. A naive algorithm addressing this issue is provided in algorithm 3. SIB1 Messages will be tagged with information regarding the periodicity information they carry. 0 for end of transmission, 1 for scheduling SIB10 messages, and 2 for scheduling SIB11 messages, 3 if case both SIB10 and SIB11 messages are scheduled before next SIB1 message.

4.8.2 SIB11 additional data

The constraints about SIB11 data (number of segment for each message) have been mentioned in section 2.4. The first \(numberOfSegments\) messages must be in sequence. This constraint can be expressed by an external function taking an array of decision variables \(x\) and an integer \(m\) being the maximum number in the sequence. First, an array of boolean variables \(g\) (of the length of \(x\)) is created and constrained: \(g\(_i\) is true if \(x\(_i\) = x\(_{i-1}\) + 1.

\[
\text{for all } i \text{ in } [1, |x|[: \quad g\(_i\) \Leftrightarrow x\(_i\) = x\(_{i-1}\) + 1
\]

Then a second array of boolean variables \(z\) (still of the length of \(x\)) is created and constrained; specifying that if \(z\(_i\) is true \(, then \(x\(_i\) must be set to zero.

\[
\text{for all } i \text{ in } [1, |x|[: \quad z\(_i\) \Leftrightarrow x\(_i\) = 0 \land x\(_{i-1}\) = m - 1
\]
The final constraint states that for a certain \( i \), \( g_i \lor z_i \) must hold.

\[
\text{for all } i \text{ in } [0, |x|]: \\
g_i \lor z_i = \text{true}
\]

Fortunately, Gecode’s MiniModel [4, Section 7] provides an easier way to formulate this constraint, avoiding explicit creation of arrays \( g \) and \( z \).

**Experiments.** Experiments were run under ArchLinux 3.14.4-1-ARCH (64 bits) on an Intel Core i3 M330 of 2.13GHz with an 3072KB L2 Cache and a 3GB RAM.

The input parameters for each type of message, and the replacement model are the ones given in section 4.7. The timestamps of notifications are generated randomly.

Figure 4.2 gives the average runtime and number of failures over 25 consecutive runs of the pipeline for various parameters. Figure 4.3 gives the average runtime and number of failures over 25 consecutive runs of the pipeline for given parameters.

The present experiments show that the model scales reasonably well with the number of notifications. The model simulates replacement behaviour, as well as the correct time stamps of messages.

Issues in scalability have to be mentioned. As the model deals with bounded domains in the first place, when the number of generated messages becomes too big (around 50 notifications), the number of decision variables and the limited size of the total domain may lead to unstable behaviour: The model can crash unexpectedly or fill the RAM completely. It must be mentioned that the implementation of the model is far from perfect. The model gives better performance during generation of messages for secondary notification, probably because the number of existing messages per repetition has a lower bound, which improves propagation.
<table>
<thead>
<tr>
<th>Notifications</th>
<th>Repetitions</th>
<th>Segments</th>
<th>Average Runtime</th>
<th>Average Number of failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>10</td>
<td>1</td>
<td>0.01</td>
<td>4</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>1</td>
<td>0.00</td>
<td>2</td>
</tr>
<tr>
<td>21</td>
<td>2</td>
<td>1</td>
<td>0.00</td>
<td>3</td>
</tr>
<tr>
<td>21</td>
<td>3</td>
<td>1</td>
<td>0.00</td>
<td>3</td>
</tr>
<tr>
<td>21</td>
<td>4</td>
<td>1</td>
<td>0.01</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>5</td>
<td>1</td>
<td>0.01</td>
<td>2</td>
</tr>
<tr>
<td>21</td>
<td>6</td>
<td>1</td>
<td>0.01</td>
<td>3</td>
</tr>
<tr>
<td>21</td>
<td>7</td>
<td>1</td>
<td>0.03</td>
<td>50</td>
</tr>
<tr>
<td>21</td>
<td>8</td>
<td>1</td>
<td>0.03</td>
<td>33</td>
</tr>
<tr>
<td>21</td>
<td>9</td>
<td>1</td>
<td>0.02</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>10</td>
<td>1</td>
<td>0.06</td>
<td>64</td>
</tr>
<tr>
<td>31</td>
<td>1</td>
<td>1</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>2</td>
<td>1</td>
<td>0.00</td>
<td>1</td>
</tr>
<tr>
<td>31</td>
<td>3</td>
<td>1</td>
<td>0.01</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>4</td>
<td>1</td>
<td>0.02</td>
<td>46</td>
</tr>
<tr>
<td>31</td>
<td>5</td>
<td>1</td>
<td>0.03</td>
<td>50</td>
</tr>
<tr>
<td>31</td>
<td>6</td>
<td>1</td>
<td>0.02</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>7</td>
<td>1</td>
<td>0.06</td>
<td>69</td>
</tr>
<tr>
<td>31</td>
<td>8</td>
<td>1</td>
<td>0.06</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>9</td>
<td>1</td>
<td>0.08</td>
<td>46</td>
</tr>
<tr>
<td>31</td>
<td>10</td>
<td>1</td>
<td>0.09</td>
<td>53</td>
</tr>
<tr>
<td>41</td>
<td>1</td>
<td>1</td>
<td>0.00</td>
<td>22</td>
</tr>
<tr>
<td>41</td>
<td>2</td>
<td>1</td>
<td>0.02</td>
<td>69</td>
</tr>
<tr>
<td>41</td>
<td>3</td>
<td>1</td>
<td>0.03</td>
<td>98</td>
</tr>
<tr>
<td>41</td>
<td>4</td>
<td>1</td>
<td>0.06</td>
<td>124</td>
</tr>
<tr>
<td>41</td>
<td>5</td>
<td>1</td>
<td>0.04</td>
<td>24</td>
</tr>
<tr>
<td>41</td>
<td>6</td>
<td>1</td>
<td>0.08</td>
<td>68</td>
</tr>
<tr>
<td>41</td>
<td>7</td>
<td>1</td>
<td>0.05</td>
<td>18</td>
</tr>
<tr>
<td>41</td>
<td>8</td>
<td>1</td>
<td>0.11</td>
<td>50</td>
</tr>
</tbody>
</table>

Figure 4.2: Generation of primary notification, average on 25 runs
<table>
<thead>
<tr>
<th>Notifications</th>
<th>Repetitions</th>
<th>Segments</th>
<th>Average Runtime</th>
<th>Average Number of failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>1</td>
<td>1</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>5</td>
<td>0.01</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>9</td>
<td>0.02</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>1</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>5</td>
<td>0.06</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>9</td>
<td>0.21</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>9</td>
<td>1</td>
<td>0.01</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>9</td>
<td>5</td>
<td>0.21</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>9</td>
<td>9</td>
<td>0.64</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>1</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>5</td>
<td>0.01</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>9</td>
<td>0.03</td>
<td>3</td>
</tr>
<tr>
<td>30</td>
<td>5</td>
<td>1</td>
<td>0.01</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>5</td>
<td>5</td>
<td>0.16</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>5</td>
<td>9</td>
<td>0.47</td>
<td>4</td>
</tr>
<tr>
<td>30</td>
<td>9</td>
<td>1</td>
<td>0.02</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>9</td>
<td>5</td>
<td>0.44</td>
<td>2</td>
</tr>
<tr>
<td>30</td>
<td>9</td>
<td>9</td>
<td>1.61</td>
<td>2</td>
</tr>
<tr>
<td>40</td>
<td>1</td>
<td>1</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>1</td>
<td>5</td>
<td>0.02</td>
<td>1</td>
</tr>
<tr>
<td>40</td>
<td>1</td>
<td>9</td>
<td>0.06</td>
<td>10</td>
</tr>
<tr>
<td>40</td>
<td>5</td>
<td>1</td>
<td>0.01</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>5</td>
<td>5</td>
<td>0.25</td>
<td>1</td>
</tr>
<tr>
<td>40</td>
<td>5</td>
<td>9</td>
<td>0.89</td>
<td>5</td>
</tr>
<tr>
<td>40</td>
<td>9</td>
<td>1</td>
<td>0.04</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>9</td>
<td>5</td>
<td>0.79</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 4.3: Generation of secondary notifications, average on 25 runs
Chapter 5
Conclusions

The abstract model, proposed for solving the message generation problem scales quite decently, however, the model itself, as well as the whole process used for solving this particular problem, may be criticized.

**Stochasticity.** One of the main problem of the model presented in this paper is the stochastic behaviour of telecommunication applications. The model has to simulate the fact that messages can be lost during transmission, and therefore model that only a part of them is actually received. There can be also random delays in transmission of messages.

The model had to reflect this aspect and a random number of received packets leads to many problems regarding the fact that Gecode expects fixed sized arrays. The size of array cannot be changed during search. There are many ways to deal with this problem.

- Create an array that could contain the maximum possible number of messages.
- Create an array with random size during the creation of the model itself.

The first solution is the one that has been retained in the model. The problem is that global constraints cannot be used, and therefore many of the most efficient and useful aspects of constraint programming are lost. Also, it might lead to a big number of decision variables which will not figure in the solution. This leads to additional computational cost that will be spent for nothing. The second way to solve the randomness problem is not a satisfactory solution either. First, one have to determine what is the minimal and what is the maximal numbers of messages that are received. This can lead to many errors regarding the implementation of the model, if
not dealt with carefully, the whole model might fail, or possibly lead to an
“Heisenbug”: a bug that only appears from time to time. For example, when
the random size of an array is too small or too big for the domain of variables,
failing the model immediately. This blending of Constraint Programming
with imperative programming is not an elegant way to address the problem
of stochasticity of communications.

There must be an elegant solution to this problem. Taking advantage of
the backtracking properties that Constraint Programming provides, as well
as avoiding loss in performance because of useless variables. It should be
possible to create messages if needed.

Gecode can deal with a special type of decision variable called set vari-
ables [4]. These set variables allow the user to constrain their cardinality,
which values the set variable contains, and so on. Set variables can grow
or shrink during search and they therefore seem to fit perfectly the current
problem. However, these variables do not allow constraint of one element
of the set with another, making it impossible to specify the constraints re-
relative to repetition periods, for example. Moreover, set variables do not fit
the time-line used right now in the model, because of performance issues.
Creating a set covering the time-line the model currently deals with — often
larger than 30000 milliseconds — is not fitting both time and memory usage
requirements. However, there is probably a way to deal with this problem
using set variables, in an elegant way.

Pipe-lining issues The pipe-lining approach of solving the message gen-
eration problem in this report improves scalability of the model. Each type
of message is processed at once, and filtering data between the various stages
of the model avoids abusive reification, and protects memory space.

However, this approach might become unstable because it lacks one an
interesting property of constraint programming: backtracking. If the current
assigned value to the variable is not satisfying, constraint programming can
backtrack and try another value from the domain. Since pipe-lining is here
used, it is impossible to save an instance of the model from failing if the
instance data that has been provided to that instance is corrupted. This
assumption that the data from the previous stage of the pipeline is correct
is a dangerous assumption.

Software testing. The first problem encountered during this project was
the lack of good documentation regarding the communication protocol itself.
The specification of the protocol exists, but it covers many aspects of the 4G-
LTE standard. When it comes to ETWS, a lot of assumptions are already
made regarding the experience of the reader in the topic. Moreover, the acronym cult that seems to be the rule in the field of telecommunications does not make the specification easy to understand for readers which are not already experienced with it. Understanding how the protocol works is of course crucial for the modelling process and one can only rely on the advice and validation of someone who already know how ETWS works to progress. Which leads to the most important lesson of this project:

The first step of this project should have been testing.

The testing process is often neglected by programmers who do not see the benefits of writing code that do not solve directly the problem, however, in this project, testing might have led to a number of improvement in terms of both development speed and quality of code.

Testing an existing log is a good way to get started with the discovery of the problem. Writing a verification script is extremely helpful when dealing with a problem that is not fully understood, — which was the case during this project — since it eases the understanding of what a correct solution is.

Also, since the testing process can be implemented in imperative programming, — a programming paradigm that is more popular than constraint programming — It might be easier to develop the verification of a solution, just focusing on how the protocol works, without any consideration regarding constraint programming.

If the verification of a solution can be implemented in imperative programming, its implementation can present a number of advantages regarding the modelling approach. Verifying a solution can often be defined as a sequence of predicates. Fortunately, predicates and constraints conceptually are the same thing. The only difference between a constraint and a predicate is that the predicate is a question, while the constraint is an order. A constraint is simply forcing a predicate to be true.

In that sense, this project enlightened one of the most interesting aspects of constraint programming: testing is modelling.
Bibliography


