JECDAR 0.2

Model checking refinement relations for Timed I/O Automata

Project Report

Group sv102f19

Aalborg University
Department of Computer Science

Copyright © Aalborg University 2015

Summary

The idea of this project revolves around the field of model-checking tools that are used to automate the process of verification and validation of certain properties for models defined with an automata-based language. One of the prominent examples of such tools is UPPAAL[1]. Starting from 1995 and until now it has developed into a large family of tools containing multiple branches, with UPPAAL STRATEGO[2] being the latest one.

In particular, the branch of UPPAAL called ECDAR[3] became the subject of interest. This tool operates with models that are based on the theory of Timed Input/Output Automata[4] (TIOA). Being a specification theory, it supports various features including *refinement, composition, conjunction* and *quotient*, as well as various checks including *determinism, consistency* and *implementation*.

However, all the UPPAAL tools are only available freely for academia related purposes. Therefore, the goal for the two-semester Master's project became the development of a new open source model-checking engine that operates with the same models and supports the same features as the existing ECDAR 0.10 tool. This involves familiarizing with existing theories, designing and implementing a new Java Engine for Compositional Design and Analysis of Real-time systems (JECDAR).

In this project we continue the development of JECDAR, which is the result of work done by Kiviriga, Klicius and Simionescu[5] during the 9th semester. At the start of this project the existing features of JECDAR included *refinement, composition and conjunction*, setting our goal to continue the implementation of the remaining features and checks.

Unexpectedly, soon in the semester two main issues were discovered that drastically changed the direction of the project. First of all, despite the assumed correctness of the implemented features, some state-space exploration problems were found in JECDAR during the verification of the refinement feature. This led to extensive manual testing of various models in ECDAR 0.10 to verify the correctness of

[1] Larsen et al. (1997)
[2] David et al. (2015)
[3] David et al. (2010a)
[4] David et al. (2010b)
[5] Kiviriga et al. (2019)

our assumptions regarding the outcome of queries. Second of all, we previously assumed ECDAR 0.10 to contain no issues and all the feature verification results of JECDAR were compared to ECDAR 0.10 to ensure correctness. However we have also discovered a number of inconsistencies between its implementation and theory, which forced us to investigate the issue further. The discovery, understanding and documentation of these issues became the main goal of the project, as only this way would we be able to guarantee correctness by solving these issues.

To combat the discovered issues and come up with the correct solution we propose a number of concepts, each of which is meant to solve a specific problem. Some of these concepts end up being not suitable, but become a ground for new ones, finally leading to the discovery of an approach that ensures correct refinement verification. This allows us to implement the possibility to return the non-canonical refinement relation.

In addition to that, we implement *determinism, least fixpoint consistency, full consistency* and *implementation* checks in JECDAR. To verify the correctness of all the features we use the test suite that includes a big number of corner cases which were found during the inconsistencies discovery phase. Both code coverage and branch coverage testing is performed to ensure correctness, with nearly 100% coverage.

The development done for JECDAR 0.2 is a great step towards having an open source model-checking engine with functionality similar to ECDAR 0.10, where the former does not exhibit the same inconsistencies with the theory as the latter one. Nonetheless, there is a lot of room for improvement. This could include the implementation of new features, use of abstraction techniques, performance optimization of the code, introduction of new modules supporting different automata theories and more.

Department of Computer Science
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
JECDAR 0.2

Theme:
Model checking

Project Period:
Spring Semester 2019

Project Group:
sv102f19

Participant(s):
Andrej Kiviriga
Cristina Ioana Simionescu
Napalys Kličius

Supervisor(s):
Ulrik Nyman

Page Numbers: 72

Date of Completion:
May 30, 2019

Abstract:

This report documents the continuation of the work done in order to implement the second iteration of JECDAR, or a Java Engine for Compositional Design and Analysis of Real Time Systems. This tool is proposed as an open-source alternative to ECDAR, an already existing environment that implements the theory of timed input/output automata, which is what JECDAR does as well. While the first version of JECDAR contained features like refinement, composition and conjunction, this version addresses various shortcomings in an attempt to reach correctness. Moreover, it adds features such as a determinism, consistency and implementation check. In the process, several inconsistencies between ECDAR and the theory of timed input/output automata were discovered. We document them and employ various methods to ensure our engine avoids such pitfalls.

The content of this report is freely available, but publication (with reference) may only be pursued due to agreement with the author.

Contents

Preface

This is a master thesis project which is a continuation of the pre-specialization semester project held by the same group consisting of three Computer Science (IT) students at Aalborg University.

We want to give special thanks to our supervisor Ulrik Nyman for all the in-depth consultancy provided throughout this semester. We express our appreciation for all the time dedicated to this project.

To benefit the most from this report, the reader is referred to Kiviriga et al. (2019) to learn more about the purpose of JECDAR, the theory of the models it operates on and its development state at the beginning of this project.

Aalborg University, May 30, 2019

Chapter 1

Introduction

This chapter briefly describes the scope of the project, the goals set and the development approach.

1.1 Project overview

Chapter 2 provides a detailed explanation of preliminaries that are necessary to understand the rest of the report. It includes some fundamental theory about the refinement relation, the DBM library internal encoding and an explanation of different kinds of zones. Next, in Chapter 3 we provide observations made during the testing of Ecdar 0.10 features that help us to broaden our understanding about the correct implementation of features, as well as inconsistencies of Ecdar 0.10 with the theory, which are essentially flaws in the implementation of the tool.

Afterwards, in Chapter 4 we present a number of concepts that were introduced in an attempt to solve the discovered issues in features of Jecdar, as well as avoid all the inconsistencies of Ecdar 0.10 with the theory described in Chapter 3.

Chapter 5 presents implementation details of the newly developed features and modules. This includes an XML parser module, determinism check, consistency check, implementation check, making an automaton input-enabled, returning the refinement relation and more.

Chapter 6 describes the tests that were performed to validate the correctness of previously existing and newly implemented features in Jecdar 0.2. We also describe the metrics used to verify the effectiveness of the test suite.

Finally, in Chapter 7 we conclude the project and specify potential directions for future work related to Jecdar.

1.2 Project Scope

This project is the continuation of the pre-specialization semester and the improvement of Jecdar - a new verification engine. The initial aim of this project was to focus on implementing the possibility of having nested features, which was described in Kiviriga et al. (2019) as an unimplemented feature. Afterwards, the preliminary goal was to carry on with the implementation of various new features such as Implementation, Specification, Quotient and more, to have a complete set of features similar to those in Ecdar 0.10. However, during the implementation of nested features and some thorough testing of models in Ecdar 0.10, we discovered a number of errors and inconsistencies between the theory and the existing Ecdar 0.10 tool. Moreover, some state-space exploration problems were detected in Jecdar that all together required further investigation.

1.3 Project goal

The above-mentioned discoveries forced us to drastically change the direction of the project. The primary goal becomes the investigation of various inconsistencies, solving and implementing correct solutions in Jecdar 0.2. As in the previous semester, the aim remains on correctness of the implementation and its consistency with the theory, rather than performance optimization.

Nevertheless, a number of features are planned to be implemented, such as *determinism check*, *consistency check* and *implementation check*.

1.4 Development strategy

Due to unexpectedly arisen high uncertainty in the project, we continue working agile. It is difficult to predict the outcome of our planned observations and the time needed to come up with relevant solutions. A large part of what we have been doing this semester is iteratively suggesting solutions to problems, working them out on the blackboard, manually testing and eventually implementing. We document our discoveries and observations and explain various solutions and concepts invented for solving certain problems in Jecdar.

Chapter 2

Preliminaries

This chapter presents the most fundamental and relevant theory that is necessary to understand the rest of the chapters in the report. Note that this does not include the theory behind Timed I/O Transition Systems, Timed Input/Output Automata, Implementations, Specifications and all the associated features: Refinement, Structural Composition and Logical Conjunction as it was already defined in Kiviriga et al. (2019).

2.1 Refinement rules

According to David et al. (2010b), "A notion of refinement allows to compare two specifications as well as to relate an implementation to a specification. Refinement should satisfy the following substitutability condition. If P refines Q, then it should be possible to replace Q with P in every environment and obtain an equivalent system". This completely makes sense, since refinement follows three basic rules.

A specification $S = (St^S, s_0, \Sigma^S, \rightarrow^S)$ refines a specification $T = (St^T, t_0, \Sigma^T, \rightarrow^T)$, written $S \leq T$, iff there exists a binary relation $R \subseteq St^S \times St^T$ containing $(s_0,\ t_0)$ such that for each pair of states $(s,\ t) \in R$ we have:

- whenever $t \xrightarrow{i?}{}^{T} t'$ for some $t' \in St^T$ then $s \xrightarrow{i?}{}^{S} s'$ and $(s',\ t') \in R$ for some $s' \in St^S$.
- whenever $s \xrightarrow{o!}{}^{S} s'$ for some $s' \in St^S$ then $t \xrightarrow{o!}{}^{T} t'$ and $(s',\ t') \in R$ for some $t' \in St^T$.
- whenever $s \xrightarrow{d}{}^{S} s'$ for $d \in \mathbb{R}_{\geq 0}$ then $t \xrightarrow{d}{}^{T} t'$ and $(s',\ t') \in R$ for some $t' \in St^T$.

In the rest of the report we will refer to these rules as refinement input, refinement output and refinement delay rules respectively.

2.2 Additional outputs on the left side

The definition of refinement states that the set of actions belonging to the transition system on the left side should be identical to the one on the right side. However, this restrains the transition systems, which greatly reduces possibilities such as having the composition of multiple components on the left side.

ECDAR does not seem to introduce such restrictions, thus there can be scenarios in which it does not perform in accordance with the theory.

Our implementation tries to make a compromise between the theory and ECDAR, namely by allowing the left side to have more output actions. This is particularly useful when the left side consists of a composition of multiple automata which synchronize together on certain actions that we do not necessarily require to match on the right side. In this case, we allow the right side not to have those outputs and instead treat it as if those outputs are ignored, by adding self-loops for all missing outputs.

This newly introduced signature check comes as a precondition of the refinement. If either the set of output actions of the left side differs from the one on the right or the left side does not contain all the input actions of the right side, then the refinement check fails immediately, without exploring the state space. If the preconditions are met, only then can the check be performed.

2.3 The left side is the driving side

Probably the most important observation that was made was that the left side is the dominant side in the refinement. Without this knowledge it would be impossible to make the refinement work properly. The right side of the refinement can be the driver side only for the inputs, while the left side will dominate in all other cases. Considering the fact that two out of three rules are in favor of the left side, as well as the fact mentioned in Section 2.2, which allows more outputs on the left side, one may call the left side being the "dominant" one.

2.4 Refinement feature issues

Our implementation of the refinement feature appeared to be complete and correct, as all our tests were passing and we had significantly high code coverage. However, after some in-depth analysis we discovered that in some cases the size of the explored state-space was unusually small. At this point, we started investigating the possible causes of this problem and we identified several inconsistencies and aspects that we had not been aware of to that point, which are documented in Chapter 3.

Bound	Raw	Bound	Raw
≤ -1	-1	< 0	0
< -1	-2	≤ 0	1
≤ -2	-3	< 1	2
< -2	-4	≤ 1	3
≤ -3	-5	< 2	4
< -3	-6	≤ 2	5
≤ -4	-7	< 3	6
< -4	-8	≤ 3	7
≤ -5	-9	< 4	8
< -5	-10	≤ 4	9
$\leq ...$	...	$< ...$	...

Table 2.1: Raw values of DBM

Further, we present a selection of concepts that we introduced in an attempt to achieve a correct implementation of the refinement feature.

2.5 DBM Raw Values

The knowledge that will be required to understand some of the practical concepts from Chapter 4 includes the exact understanding of how the zones are stored. As described in Kiviriga et al. (2019), zones in Jecdar are represented with the Difference Bound Matrix(DBM) data structure, using the DBM Library implementation (*Uppaal DBM Library* (2019)). Even though one may constrain the clocks of a zone with the integer values of the bounds obtained from such constraints as guards or updates, the *DBM manual* (2019) documentation of the DBM Library states that each clock constraint is internally encoded into a **raw_t** type that is used for all DBM related computations. In the rest of the report, we refer to these encoded constraints as "raw" values. The actual bounds are referred to as "actual" values or bounds.

The main purpose of the encoding is to be able to tell the difference between the different strictness of a given bound or constraint (ex. ≤ 5 and < 5). Note that the constraints defining the lower bound of an interval, such as $x \geq 3$, will end up being stored as negative constraints (-3) due to having to flip the inequality sign to match the $<$ or $\leq$ inequality sign required by the DBM. Table 2.1 demonstrates the correspondence between a constraint and the raw value associated with it and is just a small excerpt of bounds from the huge set of possible values. Listing 2.1 provides a formula used to calculate raw values from actual ones. The understanding of this relationship will further help us to understand the origin of various problems related to concepts that are yet to be explained.

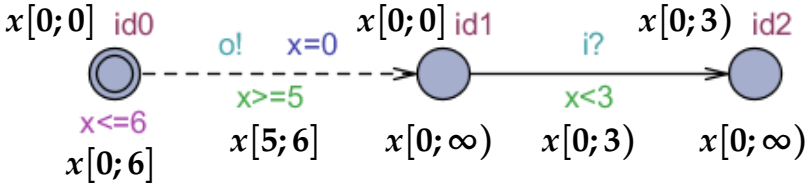


Figure 2.1: Demonstration of arrival, invariant and guard zones

Listing 2.1: Formula to calculate raw values from actual values

```
if(sign == '<') raw = actual*2
if(sign == '≤') raw = actual*2+1
```

2.6 Arrival, invariant and guard zones

In subsequent chapters of the report some of the figures of automata will contain zones generated at different times during the exploration of the state space. Each of these zones will be equally important and syntactically equivalent to the rest, however the difference in their underlying semantics will strongly influence their purpose and use in concepts. In simple words, each of these zones will contain different information related to guards, invariants and resets. In this section we introduce notation for three types of zones: **arrival zone, invariant zone** and **guard zone**.

Note that for simplicity and readability reasons we choose to represent the zones from the figures as intervals instead. However, if necessary in further examples these zones will be represented as zones instead. Additionally, only necessary information (zones, intervals, etc.) will be shown in figures in later sections to ensure readability.

Figure 2.1 shows an example of an automaton with one clock **x** and is enriched with all three kinds of zones. The zones shown above the location (mostly top-left) are referred to, in this report, as *arrival zones*. They show what values clock **x** can evaluate to when one arrives at a certain location. Since location **id0** is an initial location, it comes naturally that clock **x** can only have one possible clock valuation - 0.

Zones referred to as *invariant zones* are the ones normally depicted below the location. As a rule, the invariant zone is derived from the arrival zone by performing the delay operation followed by the application of all invariants belonging to the corresponding location. The invariant zone shown under location **id0** provides information about how long one can stay in the same location by delaying (up to **6** time units).

The last zone to be introduced - *guard zone*, refers to the zone representing all

possible clock valuations during which an edge can be taken. The guard zone is derived from the invariant zone by applying all guards associated with that edge. For example, the edge from **id0** to **id1** can only be taken with clock **x** valuations belonging to an interval from **5** to **6**.

The arrival zone of the target of an edge is then derived from the guard zone of that edge by applying all associated resets. Therefore, the arrival zone at location **id1** is shaped by the reset on the edge leading to that location, whereas the arrival zone at location **id2** is equivalent to the guard zone of the edge leading to that location due to the absence of resets.

Even though many of these zones may seem obvious and intuitively derivable from the image of an automaton, in the code of Jecdar these zones are the only source of information during the computation of Refinement, Composition and Conjunction and therefore all of the further algorithms operate with the presented zones.

Chapter 3

Observations and inconsistencies

This chapter presents observations which were made about various features of Ecdar, including refinement, implementation, consistency and determinism checks. These observations help us to get a better understanding of features that we will implement. We are also going to show the inconsistencies which were found between Ecdar 0.10 and the theory and can be considered as flaws or bugs in the implementation. This is very important as all of the test cases of Jecdar are also tested in Ecdar 0.10, and therefore we must know which test cases we have to treat correctly with respect to theory. We will also explain why this kind of inconsistencies might be happening.

3.1 An automaton does not refine itself

Based on the rules mentioned in Section 2.1, the right side of the refinement has to be able to delay at least as much as the left side or more, which will always be the case in the same automaton, because of guards and invariants being the same. The same holds for outputs and inputs, since identical automata will always be able to comply with the corresponding output or input from the other side. However, if one would investigate Ecdar 0.10 in more depth they would notice that it provides the wrong result in some cases of self refinement. For example the case of Automaton **C1** from Figure 3.1 is one of major inconsistencies of Ecdar 0.10 with the theory. If one would ask Ecdar 0.10 if **C1** <= **C1**, it would return that refinement fails. However, according to the theory, self refinement should always hold. If one was to create exactly the same automaton as **C1**, namely **C2**, the results would be exactly the same.

Nevertheless, creating the same automaton as **C1**, except having an additional reset $x = 0$ on the input edge as in Figure 3.2 would result in satisfying the refinement. Unfortunately, with the knowledge we have gathered, we cannot explain why Ecdar 0.10 treats it this way.

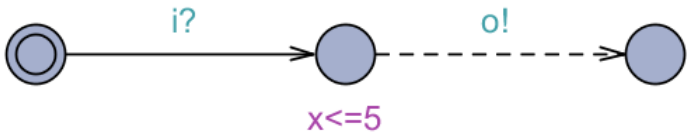


Figure 3.1: Automaton C1

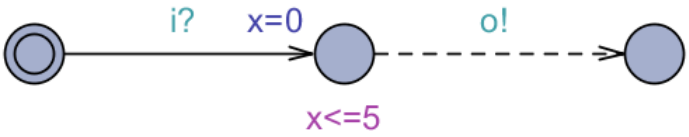


Figure 3.2: Automaton C3

3.2 No Specification as such

Based on David et al. (2010b), there exists a feature called specification, which we have described in Kiviriga et al. (2019). Throughout the year we have created multiple examples of specification checks in order to figure out how to properly implement it. However, we could not understand how exactly it works, since ECDAR 2.2 would return results which are completely incorrect according to the theory.

One of the observations that we were able to make when we started using ECDAR 0.10 was that asking for a specification check would return consistency results. We were able to see this only because ECDAR 0.10 returns an answer with a brief explanation, unlike ECDAR 2.2 which only returns true or false. This implies that ECDAR does not grant the possibility of checking for Specification.

Considering the fact that an automaton is a specification if each of its states are input-enabled and we are only working with input-enabled systems, then every automaton is a Specification. Therefore, there is no reason to have the feature Specification check as such.

3.3 Partial input-enabledness

According to David et al. (2010b), "We restrict ourselves to input-enabled systems. This makes it impossible to reach an immediate deadlock state, where a component proposes an output that cannot be captured by the other component". This implies that ECDAR converts each automaton into a corresponding automaton which is input-enabled or treats it as if it was input-enabled. However, in some cases, ECDAR fails to treat an automaton as if it was input-enabled, which makes it inconsistent with the theory. We assume that ECDAR imagines that there exists a loop of every input in every location, though it never tries to take these self loops first, which

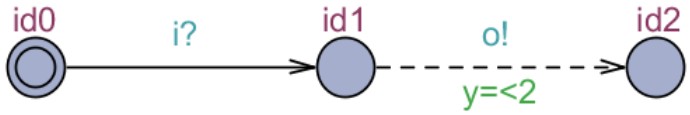


Figure 3.3: Automaton P4

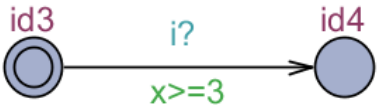


Figure 3.4: Automaton P5

cuts out some of the solutions.

For example, Figure 3.3 and Figure 3.4 represent two automata, **P4** and **P5** respectively and we deal with the case of **P4** <= **P5**. Refinement starts in state pair (**id0**, **id3**) where it checks if the right side can delay as much as the left side, which is true, then the left side checks if it has any outgoing outputs. Since there are no outputs, the right side checks for inputs, where it finds the input edge **i?**. Now one has to check if the left side can accept the same input at any given point when the right side is able to. This is true, since the right side can take that input edge at $[3;\infty]$, while the left side can do it at $[0;\infty]$. Then we arrive to the next state pair (**id0**, **id3**) with $[3;\infty]$, where we perform the delay check again, and it returns true. After the delay check one has to check for any outgoing outputs from the left side of the refinement, however given the fact that we arrive with the value of clock y being 3 or above, we are never able to take the **o!** output edge. The next step is to check if there are any input edges going from the right side and since there are none, the refinement according to ECDAR holds.

However, if one takes into consideration input-enabledness, there would be an additional path in the refinement. Figure 3.5 and Figure 3.6 provide a view of how **P4** and **P5** would look with the addition of input-enabledness. We are again concerned with the same case, **P4** <= **P5**, their state space exploration path can be seen at Figure 3.7. After checking for delays in the initial states, one can take a different path and do the self-loop on input **i?** on the right side, while on the left side taking the same edge as before, ending up in the state pair (**SP 1**) with the value of clock x being $[0;3)$. Since our arrival time includes values below 2, one can take the output edge **o!** on the left side, but the right side cannot comply and the refinement fails.

To conclude, ECDAR gives the wrong results in cases like the ones mentioned above, since it does not take into consideration input-enabledness, even though it is stated that within this field input-enabledness should be assumed.

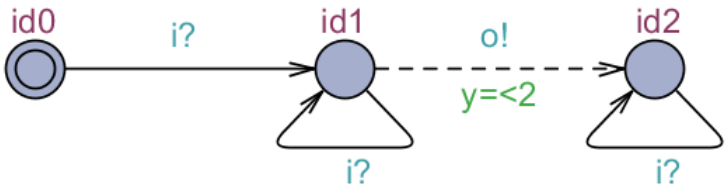


Figure 3.5: Automaton P4 input-enabled

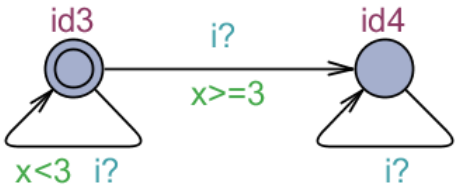


Figure 3.6: Automaton P5 input-enabled

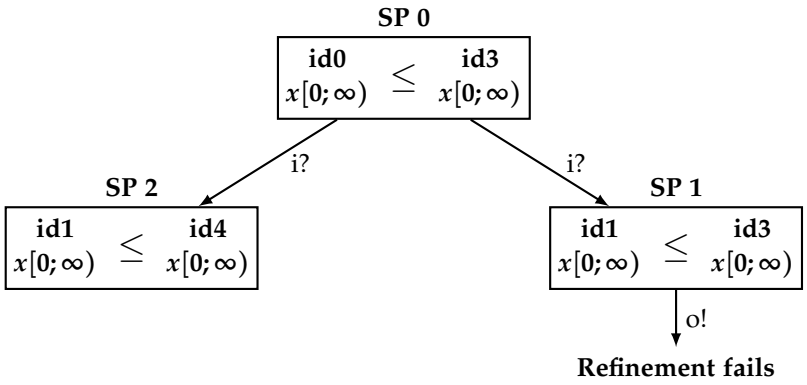


Figure 3.7: State pairs of $P4 <= P5$

Figure 3.8: Automaton T6

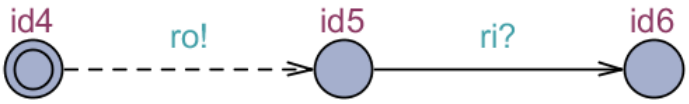


Figure 3.9: Automaton T5

3.4 Invariant bug

The case of refinement **T6** <= **T5**, whose components are illustrated in Figures 3.8 and 3.9, holds according to ECDAR 0.10. In order to understand if this is correct, one has to take a closer look at how refinement works in this case. It starts in the initial states **id1** and **id4**, where the delay check is performed with positive results, afterwards the left side outputs with an **ro!**, while the right is capable of complying to it. The delay check is once again performed on the next state pair (**id1**, **id5**) - match. Now the right side is ready to move with input edge **ri?**, however the left side cannot comply with it, thus the refinement fails.

Surprisingly, if the invariant from location **id1** or **id2** (or even both) was removed, then the tool would show that the refinement fails. It is important to remember that according to the theory, in this example the invariants should not make a difference, since the refinement would still fail due to the left side not being able to comply with an input.

3.5 Second invariant bug

An important example in our observations was the refinement (**T1** || **T2**) <= **T3**, as it revealed a similar bug to the one mentioned in Section 3.4. This case is slightly different, since depending on the invariant value or its existence, the result is different. One can see the automata that are participating in this refinement in Figures 3.11, 3.12 and 3.13. Figure 3.10 provides the state space exploration of the refinement check on such automata, according to the theory. Each box provides the view of a state pair in the refinement. Inside the state pair, one can see a simplified version of the zones, which are represented as intervals. The green color marks the value in the interval which forces the remaining zones in the corresponding state pair to shrink. Lastly, the red color indicates places where the other side of the refinement cannot comply to the leading one.

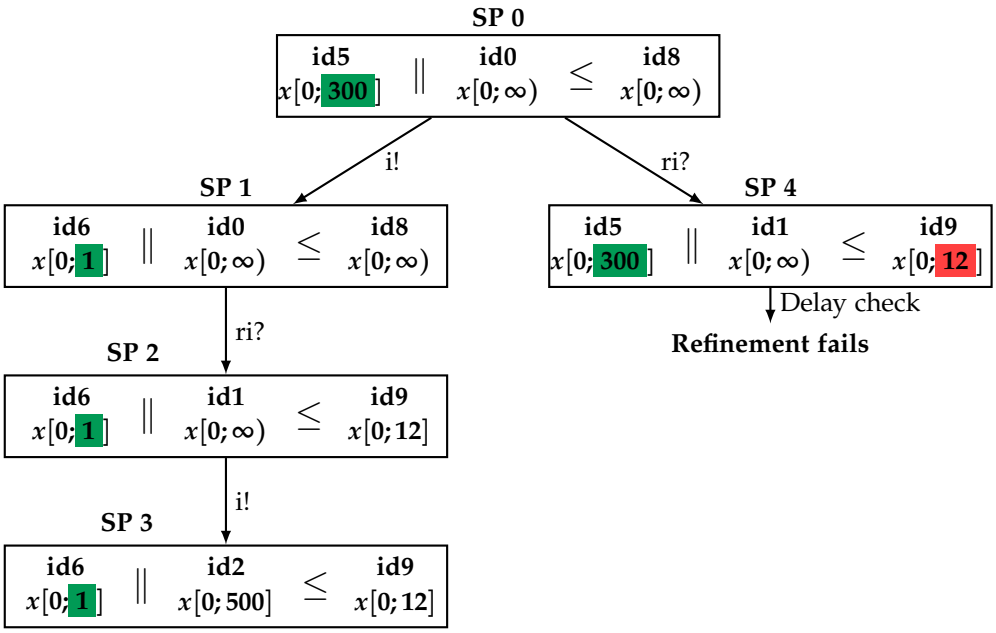


Figure 3.10: State pairs of $T1||T2 <= T3$

- (**State pair 0**) left side can delay up to 300, right side - match. Takes an **i!** edge.
- (**State pair 1**) left side can delay up to 1, right side - match. Delay check holds and takes **ri?** edge.
- (**State pair 2**) left side can delay up to 1, right side - match. Takes an **i!** self loop edge, automaton moves in composition as well with an **i?**.
- (**State pair 3**) left side can delay up to 1, right side - match. Cannot move anywhere anymore, since on the left side the invariant on **id2** is $x <= 1$ and it blocks the possibility to move, the only output edge has a guard $x >= 400$, while the right side does not have any input edges to force a movement.
- (**State pair 0**) left side can delay up to 300, right side - match. Takes an **ri?** edge - refinement move.
- (**State pair 4**) left side can delay up to 300, right side - 12, fail.

Nevertheless, performing a refinement check in ECDAR 0.10 produces a different result, which is inconsistent with the theory, namely that it holds. In order to explain why ECDAR 0.10 ends up having different results, we need to take a look at the transition going from location **id5** to **id6**. Having the invariant $x <= 1$ on location **id5** would tighten the invariant on **id5**, since one could never arrive to

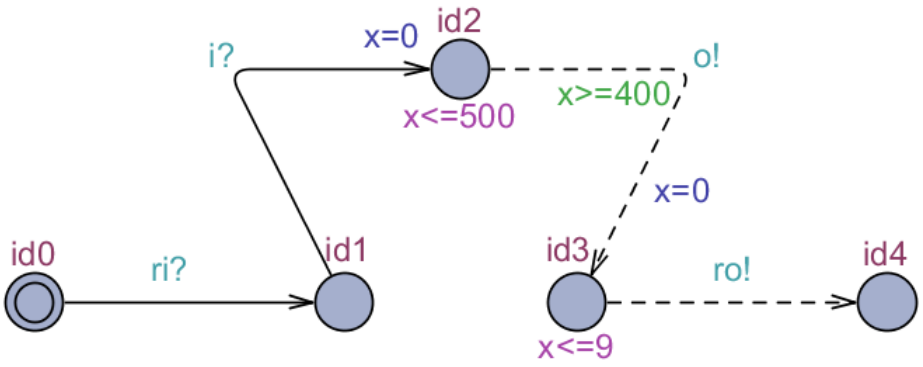


Figure 3.11: Automaton T1

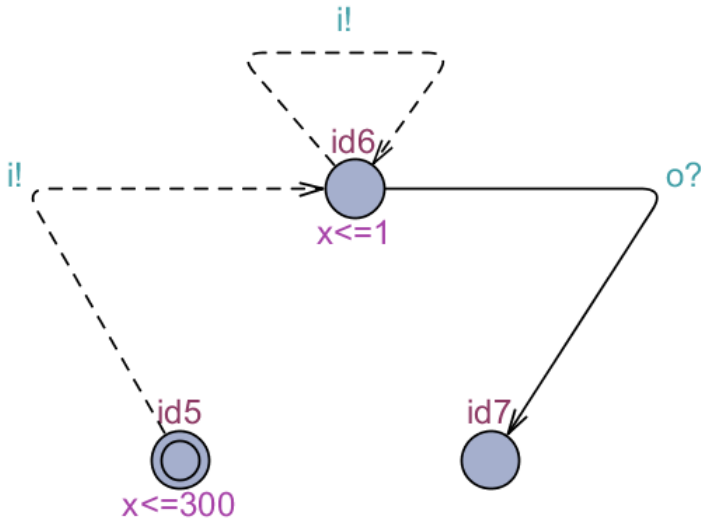


Figure 3.12: Automaton T2

id6 with values above 1. Thus the delay check in state pair ((**id1**, **id5**), **id9**) would never fail. It is important to note that this is simply a guess, since we cannot be certain why ECDAR 0.10 treats this case differently than the theory.

Another interesting phenomenon happens when removing the invariant from location **id5**. According to the theory the results should be the same, since without an invariant it would fail in the same state. However, ECDAR 0.10 outputs a different result, stating that the refinement fails. Unfortunately, we have not derived a proper explanation for why ECDAR 0.10 treats this differently than the case with the invariant, which clearly violates the delay check of the refinement.

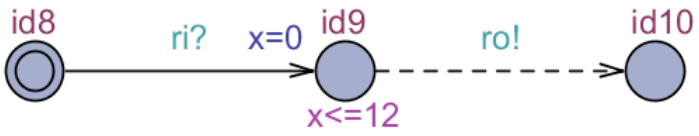


Figure 3.13: Automaton T3

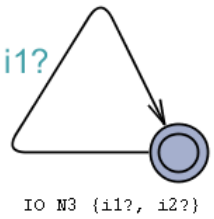


Figure 3.14: Automaton N3

3.6 Signature check

We have observed that ECDAR performs some sort of signature check. Having additional outputs on the left side, while the right side has them only in the signature but not in the component itself breaks the refinement. The same applies for the inputs, having an additional input on the right side, while the left side does not have it even in the signature, the refinement holds. On the other hand, if one would add such an input to the signature, the refinement would fail. The refinement of **N3<=N4**, whose components can be seen in Figures 3.14 and 3.15 shows the case where, according to ECDAR, having an input only in the signature of the automaton which is present on the right side leads to the failure of the refinement. However, according to the theory, all the components are input-enabled, which implies that the left side should be able to perform the "imaginary self loop" on the missing input **i2?**.

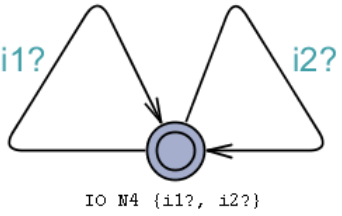


Figure 3.15: Automaton N4

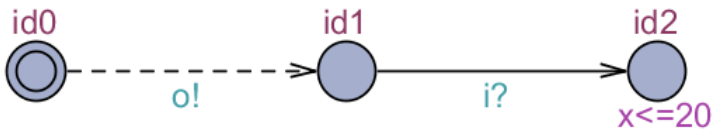


Figure 3.16: Automaton G21

3.7 Types of consistency check

In the beginning we were only aware of the existence of one type of consistency, which needs to check the whole automaton in order to make sure it is consistent. However, throughout our project we observed that there exist different types of consistencies:

- Least fixpoint consistency
- Greatest fixpoint consistency
- Full consistency

This semester we are only going to handle the least fixpoint and the full consistency. The full consistency has to check the whole automaton in order for it to return a result. On the other hand, the least fixpoint consistency can prune some of the edges in order to find the smallest part of the automaton that is consistent. It can prune all the outputs, since they are controllable, unlike inputs which can neither be controlled nor pruned. Opposite to least fixpoint consistency, the greatest fixpoint consistency finds the biggest part of an automaton, which is consistent.

Ecdar provides only one type of consistency - the least fixpoint consistency check. Consider the example in Figure 3.16, which starts in the initial location, where there is an output edge going from it. Since it is an output, we can prune the rest of the automaton and have the least fixpoint consistency. On location **id2** there is an invariant and there are no output edges going from it that could solve the problem. However, since the rest of the automaton was pruned, this does not matter anymore.

3.8 Determinism bug

An important aspect to remember is that all automata must be deterministic. If there are multiple outgoing edges from the same location with the same action, a determinism check has to be made in order to ensure that there exists only one path at any given point in time. Ecdar always performs a determinism check before running queries and it appears as though it is always able to determine correctly if an automaton is deterministic.

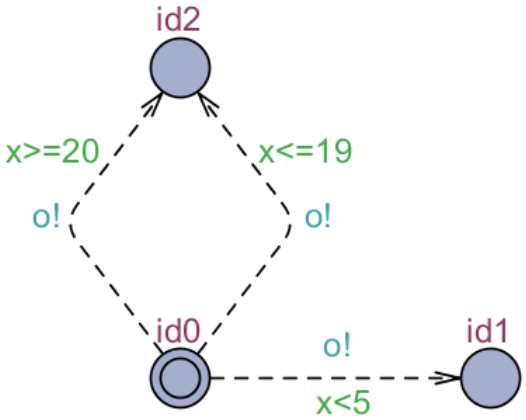


Figure 3.17: Automaton P8

In spite of this, we have succeeded to find a case where ECDAR fails at the determinism check. Consider Figure 3.17, where ECDAR states that this automaton is deterministic. In this example there are three different edges from the initial location **id0**. In the case of the two edges that lead to location **id2**, the intervals in which these edges can be traversed do not overlap, but in the case of the third edge which leads to location **id1**, the interval overlaps with one of the other edges. Thus, the determinism check should fail. It seems like in the cases where there are multiple edges going from one location to another one, they are being excluded from further determinism checks in ECDAR.

3.9 Global time correspondence

Another important observation that was made was about ECDAR's global time correspondence. Dealing with time in this field is important, so it is crucial to understand as much as possible about it. One has to keep in mind that arriving at a certain time in two automata that are in a refinement relation will always result in being at the same time in accordance with its minimum and maximum values.

In order to understand this better, one should take a look at the relation of **K5** <= **K6**, which is represented in Figures 3.18 and 3.19. The arrival time in **id1** is 0, because of the reset and the arrival time in **id4** is 5. The time correspondence is simple in this case, since in both automata we have only one value. However, in the case of the arrival times in **id2** and in **id5**, they will end up being intervals, $[0;10]$ and $[5;15]$, respectively. The actual arrival time in the previously mentioned state pair would be somewhere in the middle, but at the same point in time, given the minimum or maximum values of each interval. It is impossible to arrive in such a state pair with one of the automata being at its maximum arrival time and the other one at its minimum. The reasoning behind this is that time always moves at

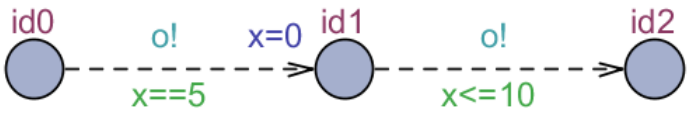


Figure 3.18: Automaton K5

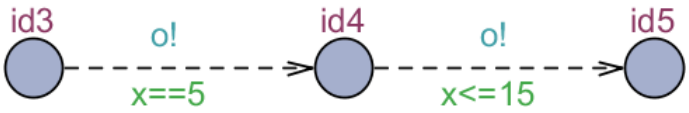


Figure 3.19: Automaton K6

the same rate and it cannot be frozen in any of the automata.

3.10 Exploration of refinement with input-enabledness

After analysing the refinement of **T9** <= **T8**, which is illustrated in Figures 3.21 and 3.20, valuable information of how refinement works in ECDAR can be derived. It starts in the initial locations **id4** and **id0**, where the delay check is satisfied. Afterwards, the right side makes a move on input edge **i1?**, while the left side is able to comply. Now the left side has only one move, which is a common input **i2?**, while the left side also has one move, namely common output **o!**. Here the refinement check is complete and ECDAR returns true, which implies that the rest of the automaton is not explored, since refinement would fail at a later point. According to the rules of refinement, the left side can force to move on outputs and the right side can do the same on inputs. The fact that ECDAR stops the exploration at this point is in accordance with the theory, since it is not stated what happens when the only possible movements are with the input from the left and with the output from the right. However, if we take into consideration input-enabledness, the situation changes. According to the theory, the refinement should fail. The state space exploration is provided in Figure 3.22 and it is performed in accordance to the theory.

The refinement check begins in state pair 0, where the right side can comply to the delay check. Both sides move with an input edge **i1?**. In state pair 1 the delays match and now we may perform a self loop on location **id1** on input **i2?**, while the

Figure 3.20: Automaton T9

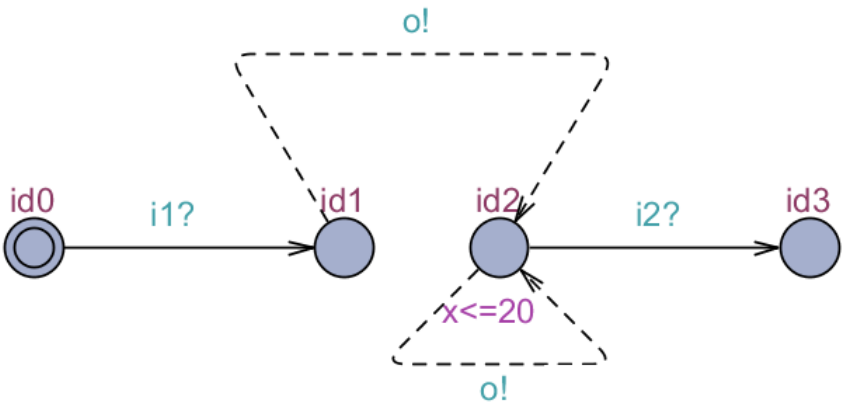


Figure 3.21: Automaton T8

left side can comply to it. Now the left side is the leading one with an output edge **o!**, which can be traversed with $x \in [0; \infty]$, while the right side cannot comply to it, since its output edge **o!** can be traversed only with $x \in [0; 20]$, thus the refinement fails.

3.11 Treatment of syncs as outputs

The refinement which is illustrated in Figure 3.23 holds in ECDAR. However, according to the theory, in this case it should not even pass the signature check. The reasoning behind this is that the left side cannot have an action **i** being an input and an output at the same time, while on the left it exists as an input. If on the right side **i** would be an output instead of an input, then the refinement would hold. This happens because syncs are treated as outputs, therefore **i** would be an output on both sides and the signature check would pass.

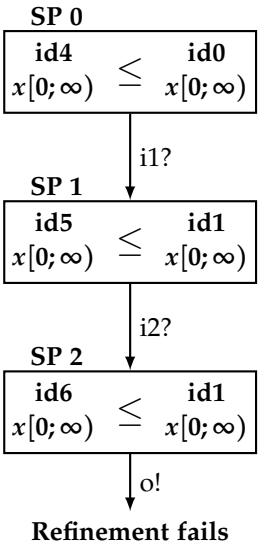


Figure 3.22: State pairs of $T9 <= T8$

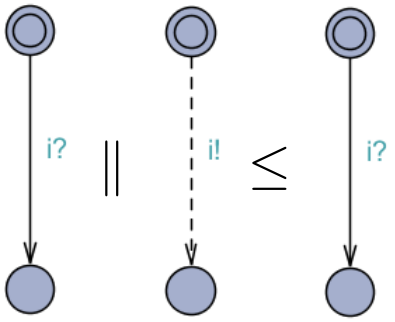


Figure 3.23: Refinement which should not pass the signature check

Chapter 4

Practical Concepts

This chapter presents practical concepts which were introduced in order to solve the refinement issues in JECDAR. Not all of the considered concepts have proven to be correct or suitable, such as concepts of arrival zone, min/max, accumulative delays and timeline. We show the problems these concepts were meant to solve and flaws of each of them. Finally we present the concept of a global zone which is the one currently used in JECDAR 0.2.

4.1 Zone unions

The data structure that we use for representing clock constraints, the DBM, is very powerful and efficient for our purposes. Until now, the operations that we needed to perform on zones(e.g., apply constraint, delay, intersection) did not cause any issues.

A problem arose when we discovered the need to add multiple zones together in order to see "the full zone" for which a certain action is enabled. DBMs "have a well-known shortcoming: they are not closed under set-union. This comes from the fact that a set represented by a DBM is convex, while the union of two convex sets is not necessarily convex", as explained in Behrmann et al. (2002).

This means that if we must perform addition or subtraction on DBMs, we must find a different data structure to store the result. The most logical approach, if we are to keep using DBMs, is to introduce the concept of zone union. On top of this, the DBM library provides support for zone unions, which simplifies the task for us.

Behrmann et al. (2002) also mentions that with a different data structure for zones, namely CDDs (Clock Difference Diagrams), performing operations such as addition or subtraction of zones would not be a problem, as these are closed under set-union. In spite of this, as long is there is no library support for CDDs, it is more convenient for us to continue using DBMs.

4.2 Absolute zone

The very first big concept that has been introduced in the project is *absolute zone*. The concept was summoned to solve a number of problems related to the verification of the availability of an edge in the context of one or more automata trying to move simultaneously to a new state in case of refinement and other features. Further we list issues that were meant to be solved by an absolute zone:

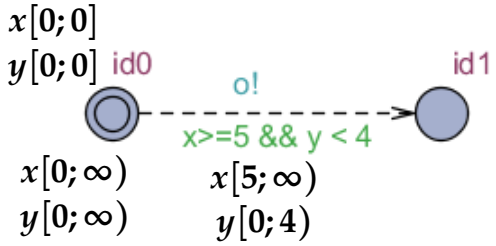
- Edge availability in an individual automaton given two or more clocks that are constrained such that an edge cannot be taken.
- Edge availability given that the constrained clocks (at least two) might end up having greatly differing possible valuations at some point of the state space exploration.
- Transition (number of edges) availability in the case of two or more automata performing a simultaneous move in such features as Refinement, Composition and Conjunction.
- Adjustment of resulting arrival zones given that two or more automata move simultaneously, resulting in arrival zones having to be constrained according to the rules of Refinement, Composition and Conjunction.

Generally, an absolute zone should be used during the call to the GetNext-Transitions() method described in Kiviriga et al. (2019), which returns transitions with all the possible moves from one state to another for an arbitrary amount of automata. It is also important to state that unlike arrival, invariant or guard zone, an absolute zone is not meant to be stored, but rather helps to properly manage the rest of the zones when taking an edge.

4.2.1 Examples

To be able to understand some of the issues better, consider the automaton in Figure 4.1. Due to the fact that **id0** is an initial location, the arrival zones state that both of the clocks **x** and **y** begin from the value **0**. In such a scenario an output edge leading to location **id1** cannot be taken because of the non-intersecting guards for clocks with identical arrival zones. However, constraining the invariant zone with the guards would result in a completely valid zone (shown to the right of the automaton), tricking one into believing that the edge can be traversed.

Next, consider a more complex refinement example of two automata excerpts in Figure 4.2, where the automaton on the left is challenged to refine the one on the right starting from locations **id0** and **id2**. The arrival zones of the two mentioned locations appear to differ to such a significant extent where it might seem from a first glance that the refinement output rule would not hold.



≤ 0	≤ -5	≤ 0
$\leq \infty$	≤ 0	$\leq \infty$
< 4	< -1	≤ 0

Figure 4.1: Unavailable edge in a single automaton and a corresponding valid guard zone

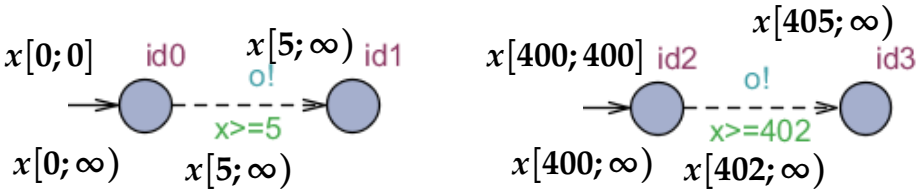


Figure 4.2: Abstract example of an available transition where left automaton refines right automaton

However, we will discover that the refinement output rule holds in this example. It is explained as follows; the automaton on the left must wait at least **5** time units before being able to issue an output, whereas the automaton on the right can always follow due to it being able to issue an output after delaying for at least two time units. Additionally, the arrival zone of the second automaton in **id3** must be adjusted to reflect that an edge was traversed only after delaying for at least **5** time units, since the right side of the refinement only follows the left side.

Figure 4.3 demonstrates two absolute zones that were built for the automata from the example in Figure 4.2. The key difference between the absolute zone and such zones as arrival, invariant or guard one, is its independence from the potentially fluctuating values of the arrival zone. The absolute zone is read as follows: the lower bound of a clock states the minimum amount of delay that is necessary to take before the edge becomes available. At least two time units must pass in order to be able to take an edge from location **id2** to **id3**, exactly which is reflected in the lower bound of the corresponding absolute zone.

Oppositely, the upper bound of the clock in the absolute zone states the maximum possible delay that can be done while the edge still remains available.

≤ 0	≤ -5
$\leq \infty$	≤ 0

≤ 0	≤ -2
$\leq \infty$	≤ 0

Figure 4.3: Absolute zones for automata from Figure 4.2

4.2.2 Implementation details

Now that we have seen some examples of the issues that the absolute zone is hoped to solve, we will take a look at the algorithm used to compute absolute zones. The function in Algorithm 4.2.1 demonstrates the main logic used to calculate absolute zones given an invariant zone and the guards of an edge that is challenged for availability.

Overall, the intent is to consider and treat respectively three cases of guards: guards that define the upper bound of the clock (Line 9), guards that define the lower bound of the clock (Line 15) and the absence of guards for certain clocks as such (Line 21). The first two cases related to different guards are treated in the first *for all* loop (Lines 3-20), which is then followed by necessary changes to the bounds of the clocks that had no guards associated with them (Lines 21-28).

First of all, we assign three variables that are essential for further computations - *constraint*, *clockUB* and *clockLB* (Lines 4-6). Note that these variables are all of type **raw_t** and correspond to the DBM library encoded constraints explained in Section 2.5. The BoundToRaw() method is used to encode the bounds of the guard into the required encoded constraint.

Due to the fact that constraining any bound of the clock will change the zone only if the applied constraint would tighten the zone, it is necessary to reset the lower bound of the clock that the guard is associated with, but only when that specific clock is handled for the first time (Lines 7-8).

Furthermore, depending on the type of the inequality symbol of the guard, the corresponding bound (upper bound or lower bound) must be calculated and then used to constrain the clock (Lines 10-14 and 16-20). Similarly, for clocks that were not constrained by any of the guards, a lower bound of the resulting zone's clock is freed, whereas its upper bound becomes the difference between the upper and lower bounds of the clock.

The reason the difference is computed with the help of the AddRawRaw() DBM method is the handy fact that the lower bound constraints result in being encoded as negative raw values. Therefore, the addition of positive *clockUB* and negative *clockLB* in Lines 13, 19 and 27 will yield the correct result.

4.2.3 Application of absolute zone

The information contained in the generated absolute zone helps to solve the issues described above. Regardless of the values of the arrival zone and other zones derived from it, we now possess "absolute" information about the time intervals when an edge can be traversed.

In the case of refinement it becomes a trivial task to check if one side of the refinement is able to follow the other one. To do so, one must compute the highest minimum delay value and the smallest maximum delay value among all clocks.

Algorithm 4.2.1 Algorithm to compute absolute zones

```
1:  function getAbsoluteZone(invZone, guards)
2:      absZone ← invariantZone
3:      for all guard in guards do
4:          constraint ← BoundToRaw(guard.value)
5:          clockLB ← invZone.clockLB
6:          clockUB ← invZone.clockUB
7:          if (firstVisit) then
8:              absZone.FreeDown(guard.clock)
9:          if (guard ∈ {<, ≤}) then
10:             if (constraint > clockUB & clockUB ≠ ∞) then
11:                 NewUB ← addRawRaw(clockUB, clockLB)
12:             else
13:                 NewUB ← addRawRaw(constraint, clockLB)
14:             absZone.ContstrainDBM(NewUB)
15:         if (guard ∈ {>, ≥}) then
16:             if (constraint − clockLB < 0) then
17:                 NewLB ← 0
18:             else
19:                 NewLB ← addRawRaw(constraint, clockLB)
20:             absZone.ContstrainDBM(NewLB)
21:     for all clock in unmodifiedClocks do
22:         clockLB ← invZone.clockLB
23:         clockUB ← invZone.clockUB
24:         if (clockLB ≠ 1) then
25:             absZone.FreeDown(clock)
26:         if (clockUB ≠ ∞) then
27:             newUB ← addRawRaw(clockUB, clockLB)
28:         absZone.ContstrainDBM(NewUB)
29:
30:     return absZone
```

The two resulting *min* and *max* values will represent the minimum delay and the maximum delay that can be taken among all clocks. After the computation is done for both sides of the refinement, it suffices to check the following two conditions:

- $min_l \geq min_r$
  Check if an edge becomes available on the right side of the refinement earlier or at the same time as on the left side

- $max_l \leq max_r$
  Check if an edge remains available on the right side for at least as long as on the left side

4.2.4 Absolute zone issues

As we will discover later on, the concept of the absolute zone appears to have certain flaws and blind spots.

First of all, historically, the concept of absolute zones appeared and was implemented before the strictness fix, described in Section 5.2. This means that any addition or subtraction of constraints was a primitive operation due to constant "weak" strictness. From math we know that the addition and subtraction of weak constraints always results in a weak constraint. Due to this fact, the algorithm of the absolute zone did not encounter any issues in its frequently occurring addition or subtraction operations on raw values.

However, together with the strictness fix new problems arose. One of the issues expected to be solved, namely the adjustment of the resulting arrival zones, became impossible to perform correctly without losing or adding solutions to the zones, neither of which are acceptable in terms of correctness.

Moreover, absolute zones do not infer any information from arrival zones and therefore a lot of vital information is lost in the process. Consider the example in Figure 4.4 where the upper automaton is challenged to refine the bottom one. It is also shown that both invariant and guard zones are absolutely identical in both cases. This means that the algorithm would produce identical absolute zones and naturally, after comparing the resulting *min* and *max* values, we deduce that the refinement holds.

However, that is not the case in this example! It is easy to show that the refinement cannot hold due to the violation of the refinement output rule, which is caused by the only difference between the two automata - the reset on the edge from **id0** to **id1**. Consider having delayed **250** at location **id0** and outputting afterwards, resulting at *id1*. The second automaton then follows according to the refinement output rule and gets to **id4**. Due to the above-mentioned reset, the upper automaton is able to delay up to **500** time units and output, whereas the bottom one, having arrived with the clock value of **250** already, will not be able to comply to the left side of the refinement.

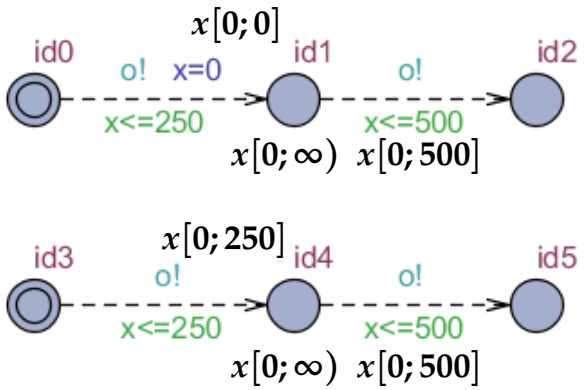


Figure 4.4: The refinement of two almost identical automata that does not hold. Upper automaton refines bottom one.

Therefore, the concept of the absolute zone alone is not enough to handle all rules of refinement correctly.

4.3 Concept of Min/Max and zone union subtraction

After realizing the failure of absolute zones to solve all the issues in the refinement relation, a new idea arose, namely Min/Max. This concept would take into consideration the very minimum of the arrival zone as well as its maximum and compare them to the corresponding guard interval. This comparison is vital when we have a reset on one of the automata, since it shows if both of the automata can delay the same amount before taking a corresponding edge.

Equation 4.1 shows the algorithm which calculates the final min and max values for the left and the right side. In this algorithm lf stands for the left side final values and rf for the right side final values, while g and a stands for guard and arrival zone respectively. After receiving the final min and max values for both of the sides, one has to perform a comparison between them in order to see if both of the sides can traverse an edge at any given value within their intervals. This comparison can be seen in Equation 4.2.

$$\begin{aligned} lfMin &= gMin - aMax \\ lfMax &= gMax - aMin \\ rfMin &= gMin - aMin \\ rfMax &= gMax - aMax \end{aligned} \tag{4.1}$$

$$\begin{aligned} lfMin &\geq rfMin \\ lfMax &\leq rfMax \end{aligned} \tag{4.2}$$

4.3.1 Examples of Min/Max

In order to better understand how the Min/Max algorithm works, it is helpful to look at a few examples. The previously mentioned refinement relation from Figure 4.4 using absolute zones fails, so this case has to be solved using the Min/Max algorithm mentioned in Equation 4.1. We are interested in locations, right before the reset happens, namely **id1** and **id4**. After the calculations of the final values, which are shown in 4.3, one can see that one of the rules is not satisfied, thus the refinement does not hold.

$$\begin{aligned} lfMin &= 0 - 0 = 0 \\ lfMax &= 500 - 0 = 500 \\ rfMin &= 0 - 0 = 0 \\ rfMax &= 250 - 0 = 250 \\ 0 &\geq 0 \\ 500 &\nleq 250 \end{aligned} \tag{4.3}$$

To ensure that the algorithm works correctly, one has to take a look at an example which is opposite to the previous one. Figure 4.5 shows a refinement relation between two automata, the upper one refining the lower one. Imagine a scenario where one would delay at location **id0** until $x = 500$ and then traverse the edge. The arrival value of **id1** would be 500, while in location **id4** - 0, due to the reset. This would break the refinement relation, since the upper automaton would be able to traverse the output edge **o!**, while the right side of refinement would not be able to follow, since it has to delay at least 250 in order to traverse the corresponding edge.

Knowing that the refinement relation does not hold, one can check if the Min/-Max algorithm would provide the same results. Once again we must look at the location that has a reset preceding it. The calculation of Min/Max from locations **id1** and **id4** can be seen in Figure 4.4. The rule, which requires $lfMin \geq rfMin$

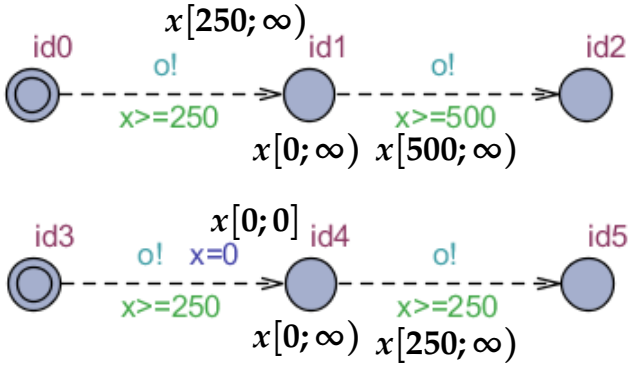


Figure 4.5: The refinement of two almost identical automata that does not hold. Upper automaton refines bottom one.

does not hold, thus the Min/Max algorithm provides the same results.

$$\begin{gathered} lfMin = 500 - \infty = 0 \\ lfMax = \infty - 250 = \infty \\ rfMin = 250 - 0 = 250 \\ rfMax = \infty - 0 = \infty \\ 0 \ngeq 250 \\ \infty \leq \infty \end{gathered} \tag{4.4}$$

4.3.2 Issues of Min/Max

This algorithm solves all the problems where there is only one outgoing edge from each of the locations. However, when we start taking into consideration examples such as in Figure 4.6, where the automaton has more than one outgoing edge, the algorithm starts providing incorrect results. The exact scenario is when the left automaton refines the right one. Both of the edges on the right side combined are capable of covering output **o!** at any given point where the left side would output, thus the refinement holds in this case. However, if one would try to place the values into the Min/Max formula, it would break. Consequently, we need a new approach for dealing with multiple outgoing edges.

Previously in Section 4.1, we have discovered the existence of DBM unions which are called Federations. When working with Federations, the subtraction method can be used in order to check if one of the Federations does not cover some part of the zone. Using this new approach one can easily deal with covering the zones of multiple edges. With this newly obtained knowledge, we have improved

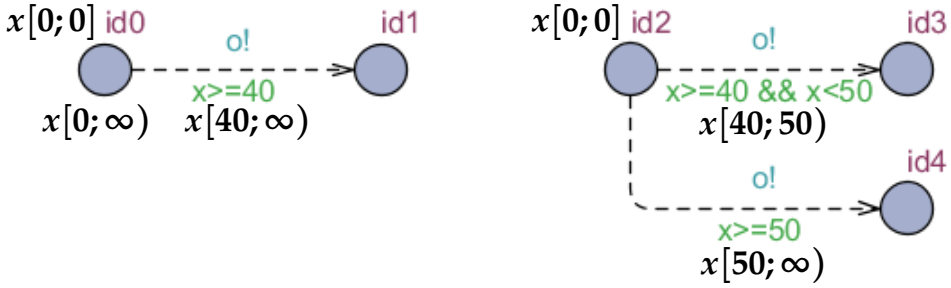


Figure 4.6: Multiple outgoing edges from a single location

the Min/Max algorithm to use subtraction of federations and slightly modified some of the formulas, which can be seen in Equation 4.5. The new Min/Max algorithm made it possible to verify if the refinement relation holds even in cases with multiple outgoing edges for the same action.

$$\begin{gathered} fminmin = gMin - aMin \\ fminmax = gMax - aMin \\ fmaxmin = gMin - aMax \\ fmaxmax = gMax - aMax \\ left[fminmin; fminmax] - right[fminmin; fminmax] = \emptyset \\ left[fmaxmin; fmaxmax] - right[fmaxmin; fmaxmax] = \emptyset \end{gathered} \tag{4.5}$$

This algorithm was supposed to work properly, but we found out that there is an issue. Math rules state that subtraction cannot be used with two inequalities of the same sign, meaning that operations such as $min - min$ bounds or $max - max$ bounds cannot be performed.

4.4 Accumulative delays

Another issue that we faced while trying to implement refinement is accumulative delays. Such concept arises when there exists a composition on one of the sides of the refinement relation. During composition, multiple automata will have internal moves, which could have different delays with resets. When a reset happens in an internal move, one must keep track of how long it was possible to delay before taking such action, thus one has to perform an addition of delays for every single internal action that has delays. In order to better understand the issue, one may take a look at Figure 4.7.

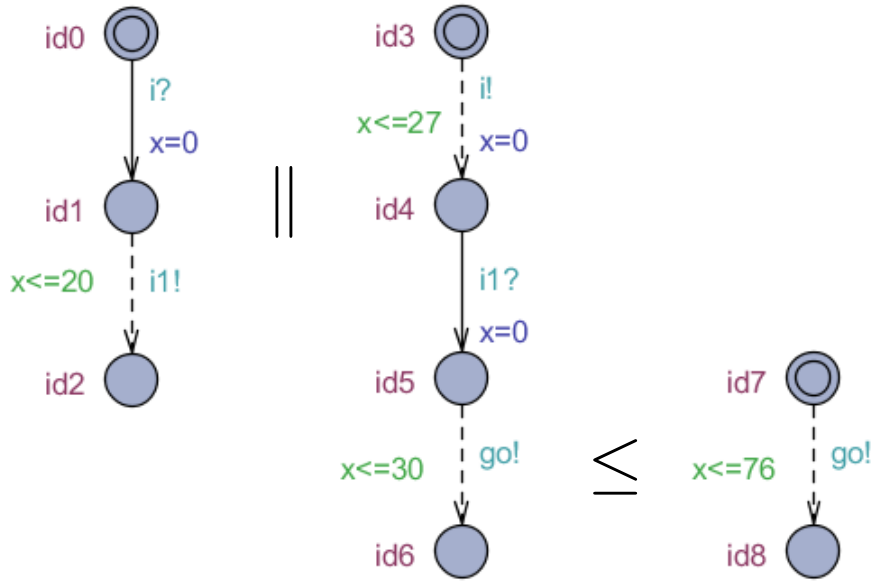


Figure 4.7: Refinement, which requires accumulative delays

If one would directly check the availability in time of the common actions in the refinement, then one could say that the refinement holds, since the right side can take the $go!$ action until $x <= 76$, while the left side requires the output edge to be available only until $x <= 30$. However, the relation must be analyzed from the very start, where multiple resets happen before the refinement move, when a common action is taken. The first move that is taken is from **id3** to **id4** with an output action $i!$, where the value of maximum possible delay, which is 27, has to be kept because clock x is being reset here. On the other side of the composition, the possible time of traversing the edge between **id0** and **id1** shrinks down from ∞ to $x <= 27$. The next composition move is being performed from **id1** to **id2** with an output edge $i1!$, where the maximum possible delay is until $x <= 20$, it shrinks down the right side's possibility to move from **id4** to **id5** from ∞. Here we have to perform the accumulation of delays, since we have a reset again. The first max delay which happened was 27, while the second one was 20, so the total is 47. Finally, one has to perform a refinement move with the common action **go!**, where the delays have to be added once more, until the next internal moves are going to happen. The right side can take the **go!** edge, as long as $x <= 30 + 47 = 77$, while the right side cannot comply to the last x value, so the refinement relation does not hold in this case.

4.4.1 Implementation details

After seeing the issue and the basic idea of how it can be solved with an example, one can take a look at the algorithm which computes the accumulation of delays. The function in Algorithm 4.4.1 demonstrates the main logic used to calculate the internal delays, given the source state and the target state.

The very first thing that is checked, in line 2, is if there was a reset, which is done by checking if the arrival min and max values are equal to zero. This check is important, because the accumulation of delays should happen only after reset, since time moves linearly and one can see the maximum delays within the guard zone. If the reset did not happen, we set the delay sum of the target state (DSum) to be equal to the guard's maximum. In line 3, we check if the guard maximum is infinity or if our current DSum is equal to infinity, then we set our target's DSum to be equal to infinity. The else statement in line 5 simply adds up the source DSum with guard's maximum and stores it in the target's DSum. This accumulation has to be performed on both sides of the refinement, since both sides can have internal moves.

Algorithm 4.4.1 Algorithm to compute accumulation of delays

```
1: function accumulateDelays(sourceState, targetState)
2:     if (aMin == 0 & aMax == 0) then
3:         if (source.DSum == INF || gMax == INF) then
4:             target.DSum ← INF;
5:         else
6:             target.DSum ← source.DSum + gMax;
7:     else
8:         target.DSum ← gMax;
```

At the end of the day, there were no significant issues while using accumulative delays. However after introducing a new concept, which is described in Section 4.6, there was no need for accumulative delays anymore. The reasoning behind this is that Global zone can handle it without any external help from such a function.

4.5 Timeline

After facing such concepts as the absolute zone, Min/Max values and not being able to compare zones of the two sides of the refinement with the help of zone union subtraction, due to the possibly differing amount of clocks on each side, it was decided to introduce a *flattening* abstraction.

The idea originated from previous discoveries: one is not able to compare zones of the left and right side of the refinement. The subtraction or intersection of the

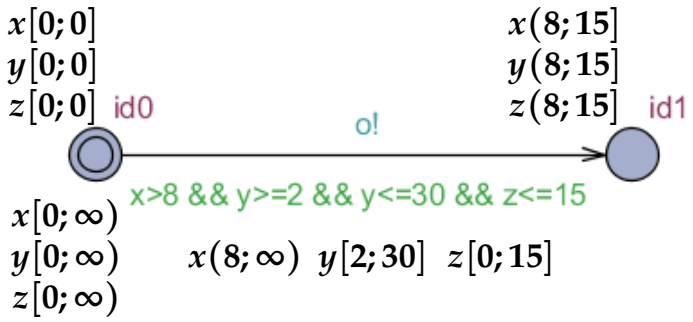


Figure 4.8: Three clocks resulting is identical arrival zones after taking an edge

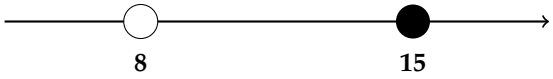


Figure 4.9: Timeline for the example in Figure 4.8

zones that is required to verify refinement rules did not appear possible to be performed, not only due to differing clock sizes of the refinement sides, but also because those clocks cannot be semantically compared (related to each other).

As a result, it is necessary to compare the relationship of a set of an arbitrary amount of clocks with another such set. To achieve that, we introduce the concept of the *timeline*, which in principle allows to flatten a multidimensional zone into a single dimension representing common results. Consider Figure 4.9, where a simplistic automaton has only one output edge, but three different clocks - x, y and z. Since location **id0** is the initial one and has no invariant assigned to it, the arrival and invariant zones of all three clocks evaluate to **0** and allow to indefinitely delay in that location respectively. The observation of the arrival zone of the location **id1** is of great interest: each of the clocks is constrained by the guards related to the rest of the clocks. This happens due to natural synchronous time progress that is identical for all of the clocks.

This observation evolved into the concept of flattening an arbitrary amount of clocks into a single "global" clock that represents all the possible clock valuations fot which an edge can be traversed. Figure 4.9 shows an example of the timeline for the automaton in Figure 4.8. Since the timeline can always be represented by one clock, we choose to draw it as an interval, where the values of the interval are derived from the "tightest" guards. The semantics of the presented timeline is as follows: in order to traverse an edge from **id0** to **id1**, more than *8* time units must pass ($x > 8$ being the tightest lower bound constraint) and the edge remains traversable until *15* ($z \leq 15$ being the tightest upper bound constraint).

It is important to note that the concept of the timeline is also supposed to consider the fact that the refinement holds only when all the possible scenarios

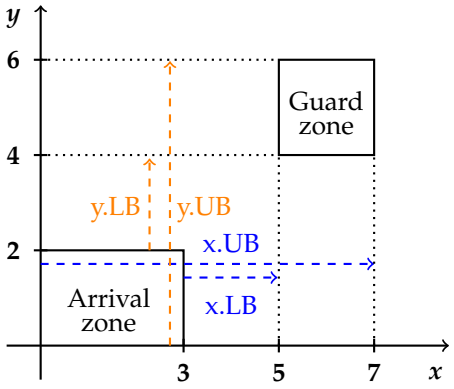


Figure 4.10: Computation of lower and upper bounds for further construction of a timeline.

of state-space exploration satisfy its rules. The interval line(timeline) must reflect all the possible valuations that are common for all the clocks when the edge can be traversed, which includes all possible valuations for clocks derived from arrival zones.

4.5.1 Implementation details

To get a better intuition of what a timeline is and how it can be computed, consider the example in Figure 4.10, where the arrival and guard zones are shown. As already mentioned, the arrival zone plays an important role in the computation of the resulting values of the timeline.

Intuitively, the lower bound (LB) for a certain clock is the difference between the LB of the guard zone and the upper bound (UB) of the arrival zone for the respective clock. On the other hand, UB is the difference between the UB of the guard zone and the LB of the arrival zone for the respective clock.

Algorithm 4.5.1 demonstrates the computational process of the timeline. The idea is to compute the LBs and UBs for each of the clocks (Lines 4 and 5), and to pick the tightest bounds among all calculated LBs and UBs (Lines 10-13). The resulting two tightest bounds (final lower bound (FLB) and final upper bound (FUB) are the ones used to construct a timeline. Note that similarly to the previous algorithms, the difference between UB and LB is computed with the help of the addition function (ADDRAWRAW), which is suitable due to the fact that the raw value of the LB is negative.

Also note that the resulting LB is expected to be a negative raw value, which is natural for all LBs. If the LB exceeds its allowed maximum value, it is set to the maximum value (Lines 6-7). This is a frequent example in cases where the arrival zone overlaps with the guard zone.

The UB also has its own minimum value, at most evaluating to **0** raw value.

If the algorithm detects UB evaluating to negative raw values, it reports failure (Lines 8,9). This might happen in cases where the guard zone is "behind" the arrival zone in time, and therefore there are no clock valuations when an edge could be traversed.

Algorithm 4.5.1 Algorithm to compute timeline

```
1:  function getAbsoluteZone(arrZone, guardZone)
2:      LB, UB, FLB = 1, FUB = Integer.MaxValue − 1;
3:      for all clock in zone do
4:          LB ← addRawRaw(arrZone.clockUB, guardZone.clockLB)
5:          UB ← addRawRaw(arrZone.clockLB, guardZone.clockUB)
6:          if (LB > 1) then
7:              LB ← 1
8:          if (UB < 1) then
9:              return null
10:         if (LB < FLB) then
11:             FLB ← LB
12:         if (UB < FUB) then
13:             FUB ← UB
14:
15:     return new Zone(1, FLB, FUB, 1)
```

4.5.2 Zone adjustment

The timeline shows us absolute clock valuations during which an edge can be traversed and helps to verify if for all the solutions of one side of the refinement, the other side is able to "follow" by traversing an edge with the same action. However not only that, but the timeline can be used to adjust zones of both sides of the refinement.

Consider the automaton example in Figure 4.11 which is the right side of the refinement, whereas the automaton of the left side of the refinement has been shown in Figure 4.8. In order to construct a proper state pair of locations **id0** and **id3** for the refinement relation it is crucial to have proper zones to ensure the correctness of further state-space exploration, if any.

The arrival zone at location **id3** in Figure 4.11 which is depicted in red is an example of an incorrect zone in case of the refinement. Such zone would be correct if the automaton was explored outside the refinement relation or if it was on the left side of it. However, the timeline in Figure 4.9 has shown that the output edge is traversed only in the interval of clock valuations from 8 (not included) until 15 (included). According to the refinement output rule, we are only interested

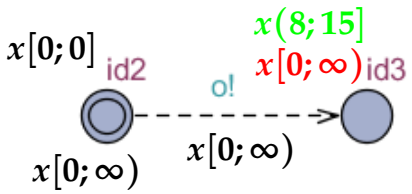


Figure 4.11: Automaton that is refined by the automaton from Figure 4.8

in exploring such clock valuations on the right side as dictated by the left side. Therefore, the correct arrival zone would have to be $x(8;15]$ (depicted in green).

Such a zone adjustment can be achieved by the following steps:

- $LB_{AZ} = LB_{IZ} + LB_{timeline}$
  To get the lower bound of the target arrival zone it is required to increase the source location invariant zone lower bound by the lower bound of the timeline.

- $UB_{AZ} = LB_{IZ} + UB_{timeline}$
  Set the upper bound to be the same as the lower bound and then increase it by the upper bound of the timeline.

4.5.3 Timeline issues

In practice, the concept of timeline appeared to be capable of solving the same problems that absolute zones and min/max concepts did, but at a much more efficient and intuitive level. Unfortunately, it was discovered that the timeline has a number of flaws that prevent this concept from providing the expected results in some corner cases.

First of all, the information about the correct strictness may be lost after constructing a timeline. The example in Figure 4.12 shows two cases where identical arrival zones and two different guard zones are used in each case to construct corresponding timelines. The resulting timelines appear to be identical, even though the strictness of the upper bound of the guards they were derived from differ. This is correct from the mathematical point of view, as the addition of two bounds, s.t. at least one of them is strict, will always result in a strict bound. However, in further computations that would result in incorrectly constructed arrival zones, as the information about exact strictness(one bound being non strict) is lost.

Secondly, we discovered that in some cases even if the timeline is correct, it does not anymore preserve the necessary information for the zone adjustment algorithm. Similarly to previous examples, Figure 4.13 demonstrates four cases of arrival and guard zone combinations with varying strictness of the bounds. In fact, even though all the timelines are identical, they are absolutely correct according to their primary purpose. Each of them states that in all the possible scenarios

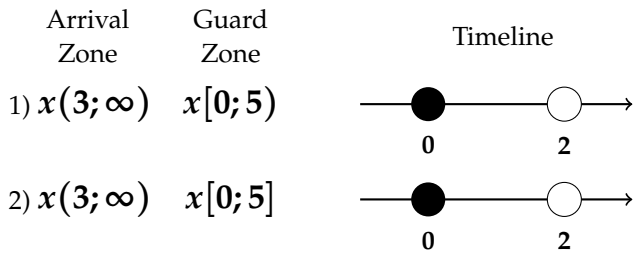


Figure 4.12: Two identical timelines for different strictness guards

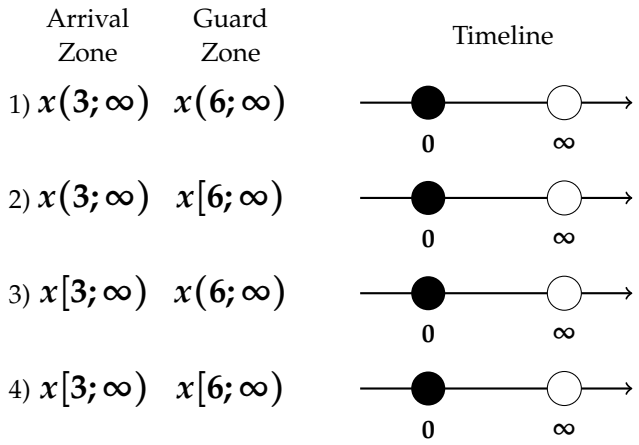


Figure 4.13: Four identical timelines for varying strictness of arrival and guard zones

of the state-space exploration joined together (which is driven by different arrival values), one might be able to traverse an edge immediately or to delay indefinitely and still be able to do so. However, such a timeline is not suitable for proper zone adjustment as it has lost information about the necessary delay in some of the cases.

To fix these issues it would perhaps suffice to to enrich the timeline to be able to preserve more information that will be necessary in the phase of zone adjustment, however a new, more powerful concept was discovered.

4.6 Global Zones

So far we have discussed a number of concepts such as arrival zone, Min/Max, accumulative delays and timeline, all of which shared similar problems - loss of information, impossibility of subtraction of opposite bound constraints and more. All of these previous concepts required a number of changes to perform correctly, however then they would become too unintuitive and cumbersome to implement.

This section presents the final concept that is used in the implementation of

JECDAR 0.2 - the one of the *global zone*. As described in Section 2.1, the refinement relation rules are applied on the pairs of states that belong to the refinement relation. Each state in the state pair represents a number of corresponding locations and contains its own zone that keeps track of all clocks from relevant automata. It means that both left and right sides of the refinement (left and right states in the state pair) existed independently from each other. Having their own zones that could not be related or compared due to their potentially differing size, proper state-space exploration required some mechanisms for zone comparisons, all of which were described in previous sections.

Moreover, clocks of different sides of the refinement cannot be semantically compared. However, as was discovered later, such a mindset caught us locked in the tunnel vision trap.

4.6.1 One common zone

In fact, throughout the entire development of all the previous concepts, it appears that we were unconsciously trying to find various mechanism allowing us to compare every clock of the left side of the refinement to every clock on the right side. The "flattening" approach of the timeline was perhaps the closest concept to achieving that. Finally, the concept of a *global* zone was born, which implies storing all clocks of the entire refinement in a single zone and thus having a single zone per refinement relation instead of two as before.

The global zone helps to easily and efficiently solve a number of problems and allows us to freely use such DBM Library operations as ISSUBSETEQ, DBMMINUSDBM and FEDMINUSFED due to the global zone being of the same dimension (size) throughout the entire refinement. Moreover, storing all the clocks in one zone also implies constantly maintaining the relationship between any pair of clocks.

4.6.2 Refinement rules verification

With the global zone, the verification of refinement rules becomes a much more intuitive task. Consider the example in Figure 4.14 where two automata are challenged to satisfy the refinement relation. The global zone, storing information about clocks of both sides of the refinement is used to verify refinement rules. For simplicity, axes **x** and **y** correspond to clocks **x** and **y** respectively.

We know that the refinement output rule challenges an automaton on the right side of the relation to follow the left side with the same common output (in this example **o!**). To verify that, the global zone of the initial location is constrained by the corresponding guard ($x \leq 4$) to obtain the zone of all the clock valuations when the output edge can be taken from **id0** to **id1** (zone on the left side). At this point, it is allowed to "cut" (constrain) the zone. Next, the resulting zone is then being constrained by the guard of the right side ($y \leq 2$). As can be seen on the

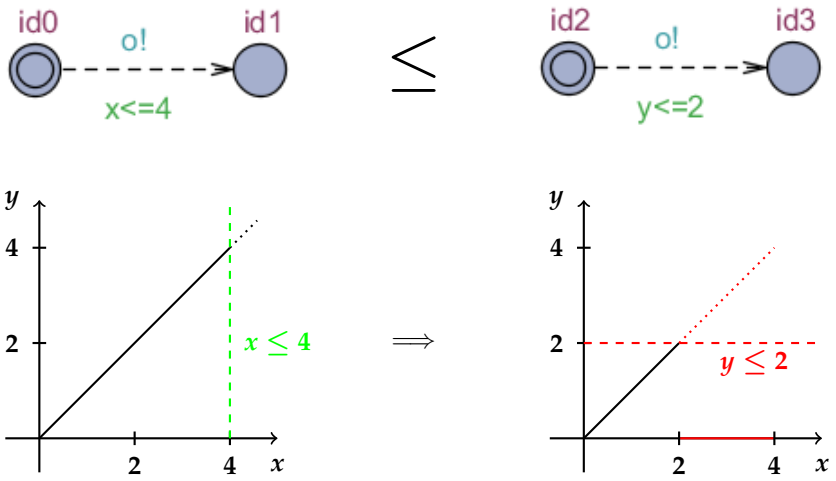


Figure 4.14: Refinement fails due to its output rule being not satisfied. Global zones illustrated after application of guards. Zone of the right side "cuts" solutions from the left side.

right diagram of the zone (Figure 4.14), the application of the guard would not only constrain the clock of the right side **y**, but would also "cut" solutions from the clock of the left side **x**. This indicates that not for all the clock valuations an output can be taken on the left side so that the right side could follow. Therefore, the refinement fails.

Algorithmically, the easiest way to discover that a zone was "cut" is to make use of the subtraction operation DbmMinusDbm provided by the DBM Library. If the result of such a method call results in anything but an empty zone, it indicates that solutions were lost. Thus, the verification of the refinement output rule requires the subtraction of the right side global zone from the left side one, whereas the refinement input rule requires the opposite.

Moreover, the concept of the global zone removes any necessity to keep track of accumulative delays, which were described in Section 4.4. This is due to the fact that zones contain not only lower and upper bounds of the clock, but also the relationship between any pair of clocks. In such cases as in Figure 4.7, if constructed properly the global zone would maintain the difference between clocks and the accumulative delays can then also be verified with the help of zone subtraction. A similar logic applies to the verification of the refinement delay rule, with a difference in subtraction being used on the global zones after the application of the corresponding invariants.

4.6.3 Extrapolation

During the verification of the refinement, the *passed* list of states is used to guarantee termination. Whenever a new state is explored, it is checked against the

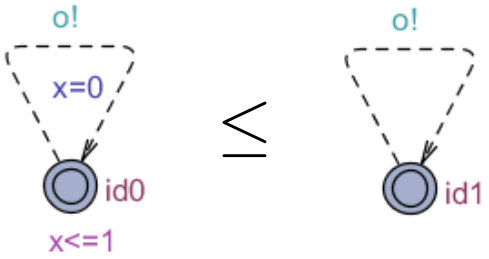


Figure 4.15: Refinement example that requires extrapolation techniques to terminate.

existing *passed* list with the following conditions: the locations of the compared states should be equal and the zone of the new state should be a subset of the state in the *passed* list. If both of these conditions hold, the newly discovered state is considered as "already explored" and is discarded. This approach used to guarantee termination when each side of the refinement had its own zone.

However, the introduction of the global zones came along with another challenge. In some cases, the state-space exploration algorithms would no longer terminate due to the zone being infinitely incremented. Figure 4.15 shows a simple example of this. Notice the invariant ($x \leq 1$) and the reset ($x = 0$) that is present on the left side of the refinement, but not on the right. During state-space exploration we would expect the algorithm to terminate as in principle no new zones or locations are discovered even after the first traversal of **o!** edge on any side. However this is not the case for the global zones.

Since global zones also preserve information about relationship between clocks during the state-space exploration, the zone would keep progressively moving upwards on the **y** axis. Thus, the termination of the algorithm would not be possible due to an infinite number of symbolic states.

To cope with this problem in Jecdar 0.2, a zone-based abstraction technique, also known as *extrapolation*, (Behrmann et al. (2006)) is used. Apart from Uppaal, extrapolation is also used in such tools as Kronos and RtSpin where it is referred to as *maximization* (Tripakis and Courcoubetis (1996)). While preserving reachability properties, extrapolation greatly reduces the state-space that needs to be explored.

Extrapolation is based on the idea that the outcome of the feature verification for automata is only affected by the changes to a clock if its value is below some certain constant. As soon as the clock has surpassed the constant its values become irrelevant, i. e. all further resulting states are identical except for clock valuations exceeding the maximum constant.

The selection of the maximal constants for clocks is an important matter as it influences the coarseness of the abstraction - the smaller the constant, the smaller state-space that has to be explored and less time and memory has to be used for

the verification of properties. The choice of the constant k involves static analysis of the automaton with the aim of finding the maximal constant appearing in either guards or invariants within the structure of the automaton. Currently JECDAR 0.2 finds one such constant for each of the clocks, which is a much coarser abstraction that using a single constant. In the example of Figure 4.15 the maximal such constant for clock $\mathbf{x}$ is $k = 1$. As soon as the upper bound of that clock in the zone is greater than k it is considered insignificant and can be replaced by a higher value being either the next highest upper bound of any another clock in the zone or an infinity if none.

The idea of extrapolation has also been taken further in a number of studies to construct coarser yet exact abstractions. For example in Behrmann et al. (2003) in addition to a constant depending on a particular clock it also depends on a particular location of the automaton. However, the current implementation of JECDAR 0.2 makes use of the extrapolation based on the maximal constants for clocks, which is a potential subject for improvement in the future work.

Chapter 5

Implementation

This Chapter presents the set of features that were implemented, while providing an explanation regarding their necessity and specific details about how they were created. This includes *determinism, consistency* and *implementation* checks, an XML parser module, various fixes and even improvements of implementation, input-enabledness, returning the refinement relation and more.

5.1 Nested features

The task that had the highest priority was to make it possible to handle nested features, a functionality that had not been supported earlier, as mentioned in Kiviriga et al. (2019). The first version of JECDAR was able to handle composition and conjunction of simple automata (e.g., Administration || Machine || Researcher or HalfAdm1 && HalfAdm2), but nesting such features was out of its scope.

We understood that handling such an issue would require changes in our data structures. A state used to be represented as a vector of locations (where each location would correspond to an automaton) and a zone. When trying to retrieve the next possible transitions that can be taken from a certain state, we would have to gather all the transitions that can be taken by each subcomponent. With a flat structure of the current location, this is hard to achieve, as we no longer know which part of the location corresponds to which transition system.

Figure 5.1 illustrates this problem. It shows a composition between another composition (A || B) and a simple automaton, C. Note that, for simplicity, only the relevant parts of the automata are displayed. The red locations represent the current location. Being in the state consisting of locations L1, L3 and L5, we would like to know what transitions can be taken with the action x. If our location was simply represented as a vector of locations, we would not know which part of it we need in order to fetch the next transitions from (A||B) and from C.

We solved this issue by refactoring our location vector into what we call a "sym-

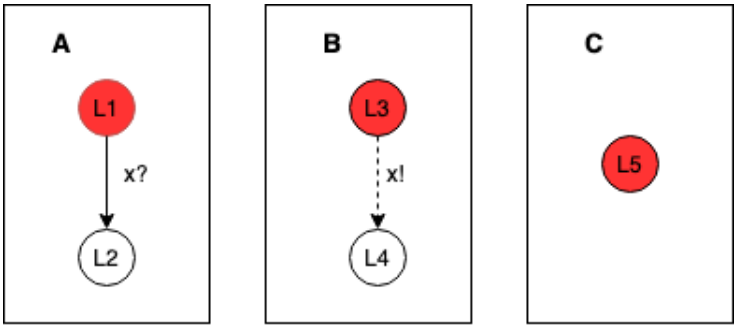


Figure 5.1: (A || B) || C

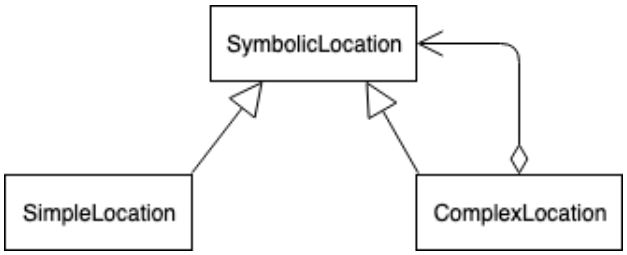


Figure 5.2: Structure of the SymbolicLocation class

bolic location". This new data type is represented by an abstract class, with two classes inheriting from it, as shown in Figure 5.2. A SymbolicLocation can be either a SimpleLocation or a ComplexLocation. A SimpleLocation is simply a wrapper for a regular Location, while a ComplexLocation can have many SymbolicLocations.

This approach allows us to store locations as tree structures than can be built in a recursive fashion. Figure 5.3 shows how the location from (A || B) || C is represented in terms of the new data structure.

Moreover, if we previously used to retrieve outgoing edges from a location, after introducing symbolic locations we discovered the need to create a corresponding abstraction for outgoing edges from a symbolic location. We gave this new concept the name of "Move", and its purpose is to connect two symbolic locations through a vector of edges. This way, we can get the moves that can be taken from each symbolic location of a certain state and then aggregate them in order to build a transition. Figure 5.4 presents the correlation between states, symbolic locations and locations and the one between transitions, moves and edges.

5.2 Addition of strictness

As explained in Kiviriga et al. (2019), since transition systems can be infinite due to an infinite amount of states given by time being continuous, we must use a sym-

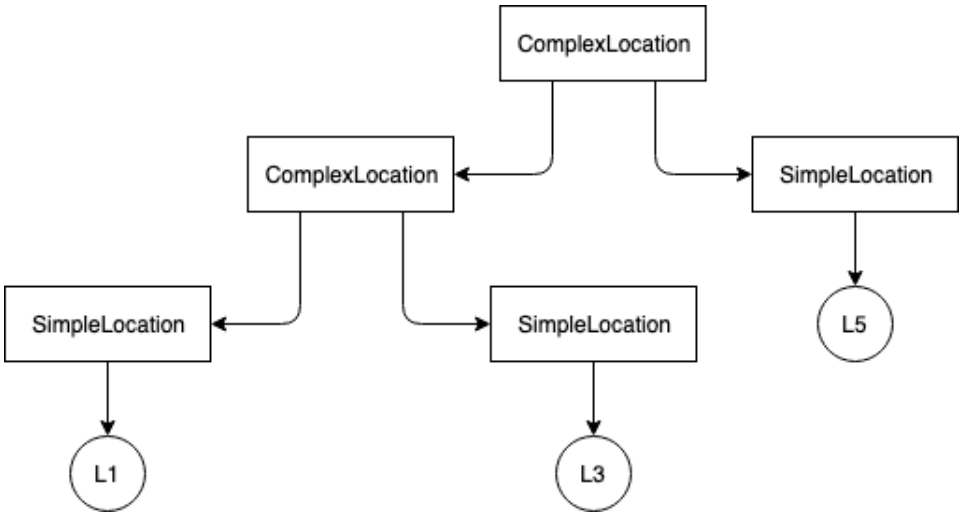


Figure 5.3: Tree representation of a location

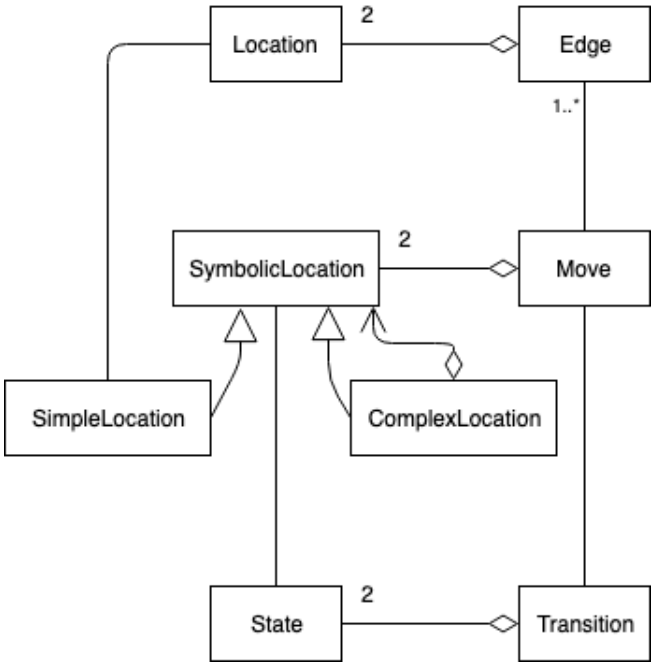


Figure 5.4: Class diagram

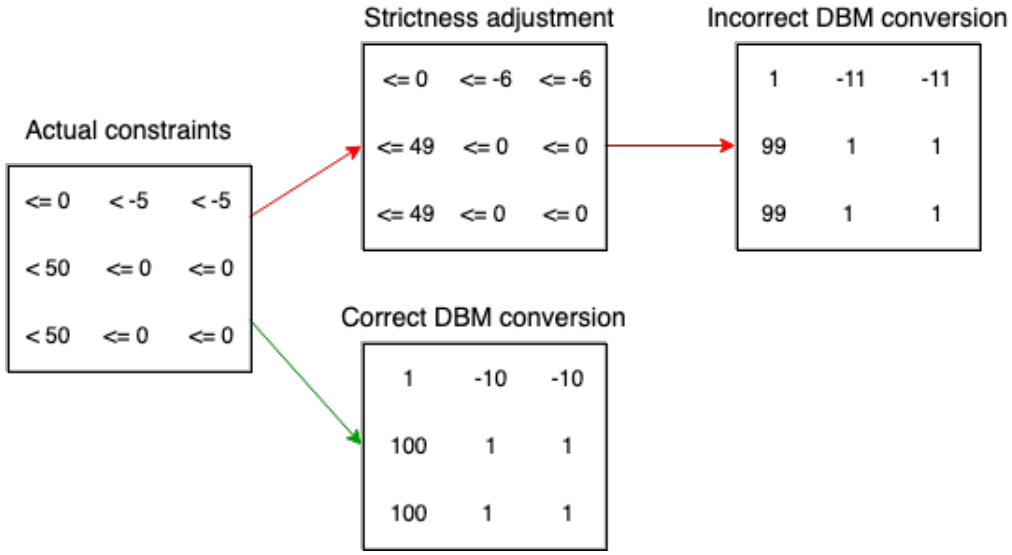


Figure 5.5: Conversion from actual DBM to internal representation

bolic representation of states in order to achieve finiteness. A common approach is to make use of zones, which contain intervals for each clock, and this is also the approach that we chose. We represent zones through DBMs (difference bound matrices), which allow us to specify the upper bound on the difference between each two clocks.

The DBM library allows us to construct such constraints and to specify their strictness ($<$ or $\leq$). The value and its strictness are encoded in each constraint. We noticed that we were building DBMs incorrectly, as by default all our constraints were non strict and we used to modify the values in order to match the strictness (e.g., $x < 7$ would be turned into $x \leq 6$, which means that the values from the interval $(6, 7)$ would be omitted).

In Figure 5.5 we show an example of an actual DBM and how the previous version of Jecdar used to adjust the values according to the strictness and then convert it into the library's internal representation. We also illustrate the correct representation that we now achieve by eliminating the adjustment and encoding each constraint according to its strictness.

5.3 XML Parser

The first version of Jecdar had a JSON parser for the models created with Ecdar 2.x. In order to automate our tests as much as possible, it was important to avoid creating test cases manually. The most suitable approach that we found was to create models in Ecdar 2.x, which can later be parsed into our internal data structures

that the tests can be run on. We would run different queries on these models and use the results to infer the expected output of each test. Our reason for choosing ECDAR 2.x was the fact that it was our intention to integrate JECDAR with the new GUI at a later point, so it made sense to use it for creating models and verifying properties of them.

After experiencing questionable results when running certain queries, we decided to check whether ECDAR 0.10 would provide us with different outputs. As expected, ECDAR 0.10 gave the same results, and that makes sense since both versions are using the same engine behind the scenes. However, ECDAR 0.10 would also display handy error messages. (e.g., the reason why a refinement holds or does not hold).

Henceforth, we made the decision to continue creating and testing models in ECDAR 0.10 and for that purpose we found it useful to create an XML parser as well.

This parser is implemented in a similar way to the JSON one. It uses a library in order to simplify the task and to reduce the chances of making mistakes. The parsing is performed by taking a specific XML file and operating on the nodes corresponding to the automata. The result is a list of automata that together make up a model.

Moreover, since we now support models in XML format as well, we introduced one additional thing. As mentioned in Kiviriga et al. (2019), we have several commands that can be used for communicating with JECDAR from the terminal. The "-rq folderPath query query..." command, which runs one or more queries, given the path of the model, has been modified with the addition of an extra flag to indicate whether the model is in JSON or in XML format. The reason for doing this is so that the controller can choose which parser it should use. In order to run such a command, "-rq" must be typed, followed by either "-json" or "-xml", followed by the actual queries.

5.4 Determinism check

According to David et al. (2010b), the specification theory was defined to be applied on deterministic TIOTS, where the following applies: for all $a \in \Sigma \cup \mathbb{R}_{\geq 0}$ whenever $s \xrightarrow{a}^{S} s'$ and $s \xrightarrow{a}^{S} s''$ we have $s' = s''$. Without this property, one would not be able to tell which edge has to be taken for an action. Therefore, determinism must be ensured for each automaton and is considered an implicit prerequisite for any further application of features on that automaton.

To check for the determinism property of an automaton, one must ensure there does not exist a non-deterministic choice on an action (edge) at any location of the automaton. The Algorithm 5.4.1 outlines the general implementation design for the determinism check.

Algorithm 5.4.1 Algorithm to verify determinism of automaton

```
1:  function isDeterministic
2:      Passed ← {}
3:      Waiting ← getInitialState()
4:      while (Waiting.hasNext) do
5:          state ← Waiting.pop()
6:          passed.add(state)
7:          for all action in actions do
8:              trans ← GetNextTransitions(state, action)
9:              if (CheckMovesOverlap(trans)) then
10:                 return false
11:             else
12:                 targets ← (tran :: getTarget) ∉ passed
13:                 add targets to Waiting
14:
15:     return true
```

The idea is to check every reachable location of the automaton, but do that only once. For that, *passed* and *waiting* lists are used. Then for each location in the *waiting* list the following applies: a) the location is moved to the *passed* list, b) for each action present in the signature of the automaton, the possible transitions are acquired, c) actions are checked for determinism (Line 9). As soon as at least a single instance of non-determinism is encountered the algorithm reports determinism to have failed (Line 10).

Algorithm 5.4.2 shows the logic behind the CheckMovesOverlap method. The idea behind the verification of an arbitrary amount of transitions for their "overlapping" lies in ensuring there is no intersection between all possible combinations of two transitions.

In principle, a single existing transition cannot create non-determinism (Lines 2-3). For a larger amount of transitions, the algorithm takes a look at every possible pair of transitions without considering the same pair more than once. Note that if a considered pair of transitions leads to the same target location and the updates (resets) of those two transitions are identical it should be considered a deterministic pair and not checked for intersection (Lines 6-8). In other words, even if two transitions lead to the same location, but their updates (resets) differ, they may create a non-deterministic choice in terms of zones for clocks if these transitions overlap.

Next, for each pair the intersection of zones is checked and if one exists then the algorithm reports non-determinism. This algorithm eliminates the problem present in Ecdar 0.10 which was described in Section 3.8.

Algorithm 5.4.2 Algorithm to check if transitions overlap

```
1:  function CheckMovesOverlap(trans)
2:      if (trans.size < 2) then
3:          return false;
4:      for (i ← 0; i < trans.size; i++) do
5:          for (j ← i + 1; i < trans.size; i++) do
6:              if (trans[i].targetLoc = trans[j].targetLoc) then
7:                  if (trans[i].hasEqualUpdates(trans[j]) then
8:                      continue
9:              zone1 ← trans[i].targetLoc.invZone
10:             zone2 ← trans[j].targetLoc.invZone
11:             if (intersect(zone1, zone2)) then
12:                 return true
13:
14:     return false
```

5.5 Consistency check

The determinism check is a prerequisite for the next feature implemented - consistency check. In fact, all automata must first pass the determinism check before being eligible for any further computations.

Algorithm 5.5.1 is suitable for checking two types of consistency: least fixpoint consistency and full consistency, both of which were mentioned in Section 3.7. The type of consistency to be checked is determined by the boolean variable *canPrune* that is passed as an argument to the CheckConsistency method together with the *state* from which the consistency check should begin, normally being the initial state.

This algorithm has a recursive nature and avoids loops in the structure of the automaton by maintaining the list of *passed* states. Any state that is checked for consistency will either be added to the *passed* list or return *true* if the state has been already explored (Lines 2-4).

Lines 5 to 9 handle all the outgoing input edges from the current state. As mentioned in Section 3.7, inputs cannot be assumed to never be received as they depend on the unpredictable environment. Therefore, regardless of the ability to prune inconsistent parts of the automaton or not, the targets of those input edges must always be checked for consistency. If the target state of at least one input edge appears to be inconsistent, the entire check of consistency returns *false* (Lines 8-9).

Interestingly, with the least fixpoint consistency it suffices to find the earliest state that satisfies the property of independent progress and simply prune the rest.

Algorithm 5.5.1 Algorithm to check consistency

```
1:  function CheckConsistency(state, canPrune)
2:      if (state ∈ passed) then
3:          return true;
4:      add state to passed
5:      for all action in inputs do
6:          trans ← GetNextTransitions(state, action)
7:          for all tran in trans do
8:              if (¬CheckConsistency(tran.Target, canPrune)) then
9:                  return false;
10:
11:     if (canPrune and state.CanDelayIndefinitely) then
12:         return true
13:
14:     for all action in outputs do
15:         trans ← GetNextTransitions(state, action)
16:         for all tran in trans do
17:             isConsistent ← CheckConsistency(state, canPrune)
18:             if (consistent and canPrune) then
19:                 return true
20:             if (¬consistent and ¬canPrune) then
21:                 return false
22:
23:     if (¬canPrune) then
24:         if (outputExisted) then
25:             return true
26:         else
27:             return state.CanDelayIndefinitely
28:
29:     return false
```

In such cases, when the consistency check is allowed to prune, lines 11 and 12 are executed. The algorithm returns *true* if the independent progress property is satisfied by the state being able to delay indefinitely.

If the state was not able to delay indefinitely or the pruning was not allowed, the algorithm proceeds to checking all the outgoing output edges. Each outgoing edge is checked for consistency by performing a recursive call and the boolean result is stored in the *isConsistent* variable (Line 17). In the case of being allowed to prune states, having found at least a single consistent state suffices for the entire consistency check to return *true* (Lines 18-19). On the other hand, not being able to prune states and having found at least one inconsistent state reached by an output edge means the consistency check is not satisfied (Lines 20-21).

After having checked both input and output edges of the state there is some logic left for cases of computing full consistency. If an output edge existed at that state and we have not found any inconsistent states reached by any edges, the algorithm can safely report the automaton being consistent(Lines 24-25). However, if no outputs existed, the only way to ensure the independent progress property becomes the possibility of the state to delay indefinitely (Lines 26-27).

Finally, if by the very end of the algorithm none of the return statements were triggered, it means that no states managed to satisfy the property of independent progress and, therefore, the algorithm terminates.

5.6 Implementation feature

The feature of verifying whether an automaton is an *Implementation* is the next step after the consistency check. In general, the implementation feature is "stricter" than the consistency check by one extra property: in addition to determinism and consistency, the automaton has to ensure the property of output urgency.

Since the algorithms for the determinism and consistency properties have already been demonstrated in previous sections, it is the output urgency property that is of interest. The verification of this property is shown in Algorithm 5.6.1. The prerequisite for this algorithm is to be applied on an automaton that is already known to be deterministic and fully consistent.

Similarly to the determinism check, this algorithm must consider every state only once, which is done with the help of the *waiting* and *passed* lists. While iterating through all the states, only the edges that are of an output action matter. For each of such edges we then check if the output urgency property is preserved. This is done with the help of the OutputsAreUrgent method that ensures that there is no possibility to delay, but only to output immediately.

Algorithm 5.6.1 Algorithm to verify the implementation property

```
1:  function isImplementation
2:      Passed ← {}
3:      Waiting ← getInitialState()
4:      while (Waiting.hasNext) do
5:          state ← Waiting.pop()
6:          passed.add(state)
7:          for all action in actions do
8:              trans ← GetNextTransitions(state, action)
9:              if (¬trans.isEmpty() and action ∈ outputs) then
10:                 if (¬OutputsAreUrgent(trans)) then
11:                     return false
12:             targets ← (tran :: getTarget) ∉ passed
13:             add targets to Waiting
14:
15:     return true
```

5.7 Federations and operations

In the DBM library, there exists a data type that is used for representing unions of zones, and that is the Federation. Internally, this is simply an abstraction over a list of DBMs which is stored as two-dimensional arrays, where each array contains values of a certain DBM and all of the arrays (DBMs) must be of the same size.

Federations provide a wide selection of operations that can be applied on them, but the most interesting ones for our purposes are the ability to subtract a federation from a DBM, a DBM from a federation and a federation from a federation.

In order to call the DBM library, which is written in C/C++ from our Java code, we followed the same approach as before. We made use of the Java Native Interface, which made it very easy given that it was already set up, so we were able to add new methods quickly. The only aspect that made it challenging was to implement a mechanism for conversion between C++ objects and Java objects (and vice versa). Whenever we pass a federation to the library, we must convert it from Java to C++ and whenever we want to return a federation generated by the library, we must convert it from C++ to Java.

Moreover, we found it relevant to create our own data type for a Federation, in order to achieve a higher level of abstraction. Figure 5.6 illustrates, side by side, the internal representation of the Federation type in Java and in C++.

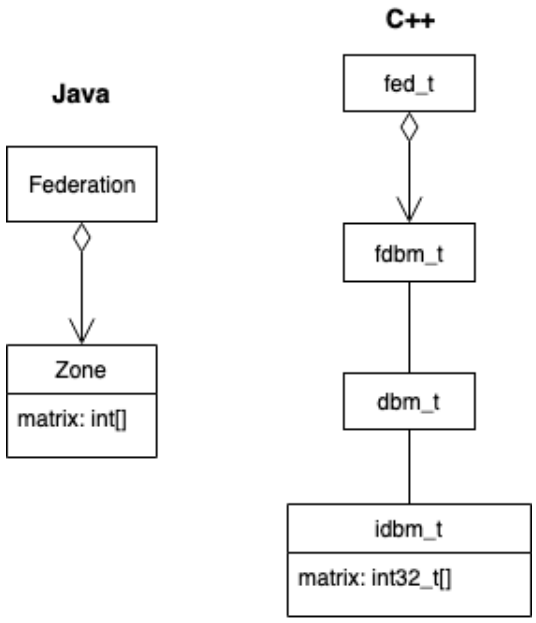


Figure 5.6: Java vs C++ representation of a federation

5.8 Using federations to detect missing intervals

When performing a refinement check, we must check for the input, output and delay rule. Since refinement is a binary relation, this must be done for each of the pairs belonging to the relation.

Let (s, t) be the pair consisting of the initial states of the transition systems derived from S and T, shown in Figure 5.7. Since we have an output transition from s defined for valuations of x in the interval [20, 50], the same should hold for t in order for refinement to hold. However, in the case of t, the output transition is defined only for $[20, 30] \cup [40, 50]$, which means that we need a mechanism for determining that the (30, 40) interval is missing.

One of our first ideas was to check the minimum and the maximum value for which the transition is enabled. This approach would fail, as the missing interval would not be caught.

An approach that would catch missing intervals is to use the subtraction of federations. This involves building a federation with zones corresponding to all edges of a given action for one of the sides and then doing the same thing for the other side. Depending on whether it is an input or an output action, we subtract one federation from the other and if the resulting federation is not empty, then we can conclude that the rule is violated.

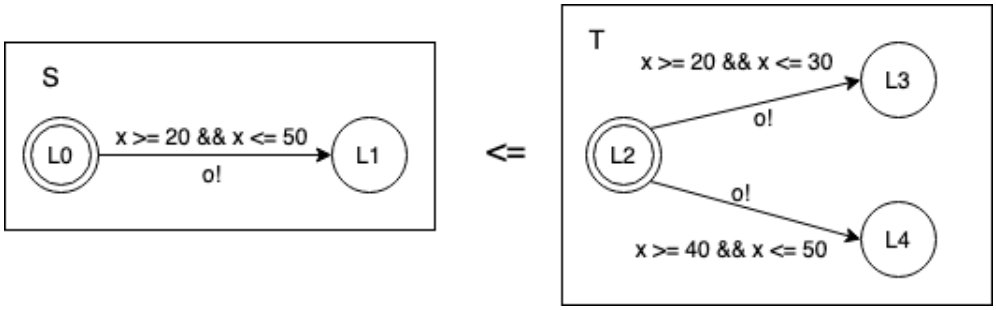


Figure 5.7: S does not refine T

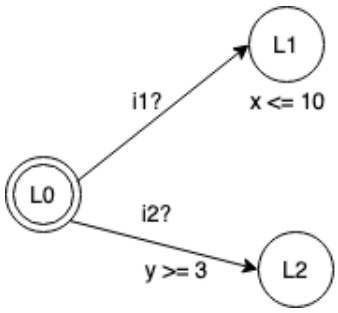


Figure 5.8: Automaton resulting in transition system that is not a specification

5.9 Input-enabledness

After concluding that each transition system should be a specification by default, we realized the need to treat each state of a transition system as if it was input-enabled.

An automaton like the one in Figure 5.8 derives a transition system that is not a specification, as not all states are input-enabled. State (L0, $\{x \in [0, \infty), y \in [0, \infty)\}$) can only accept input i1 for values of x and y lower than 10 (included) and input i2 can be taken only for values of x and y higher than 3 (included). Moreover, the states corresponding to locations L1 and L2 do not accept any inputs.

If we were to modify the previous automaton so that it would derive a specification, it would look like the one in Figure 5.9. This automaton has all the missing self-loops added to each location, in order to ensure input-enabledness.

We discovered two ways of tackling the issue of input-enabledness: either to add the missing input transitions for each state whenever we try to fetch the next transitions from that state, or to perform the operation of adding the missing input edges to each automaton the moment that it is constructed.

While the former is somewhat easier to implement, the disadvantage is that the action would be repeated each time the next transitions for a state are requested. The latter, on the other hand, is an operation that only needs to be done once and

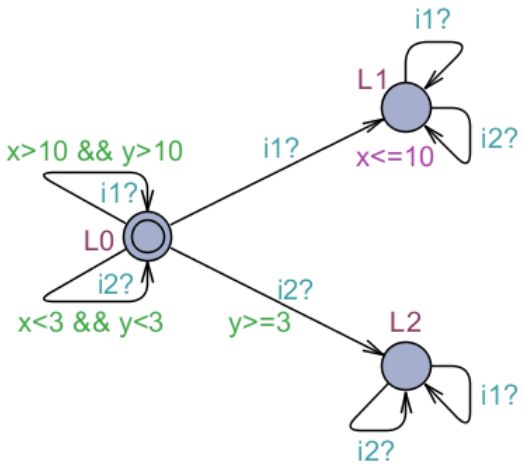


Figure 5.9: Automaton resulting in transition system that is a specification

this was a strong reason to choose it. Moreover, to simplify this task, we also opted for adding the invariant of the target location to each edge, as this makes it easier to understand the zone for which a certain input can be received and then derive the missing zones. In case there are resets on the edge, this step is skipped, as the invariant will always be satisfied.

Algorithm 5.9.1 Apply input-enabledness function

```
1: function ApplyInputEnabledness
2:     for all loc in locations do
3:         zone ← initializeZone()
4:
5:         for all invariant in loc.Invariants do
6:             zone.buildConstraintsForGuard(invariant)
7:         fullFederation ← {zone}
8:
9:         for all input in inputs do
10:            inputEdges ← getEdgesFromLocation(loc, input)
11:            zones ← {}
12:            for all edge in inputEdges do
13:                guardZone ← zone
14:                for all guard in edge.Guards do
15:                    guardZone.buildConstraintsForGuard(guard)
16:                add guardZone to zones
17:
18:            federation ← zones
19:            resultFederation ← fullFederation − federation
20:
21:            for all zone in edgeZones do
22:                newEdge ← (loc, loc, input, buildGuardsFromZone(edgeZone))
23:                add newEdge to edges
```

Our implementation of the ApplyInputEnabledness() function is shown in Algorithm 5.9.1. It starts out by iterating over all locations of an automaton, building a zone with the help of the invariants and using it to construct the federation belonging to that state. Then it looks at all the input edges corresponding to a single action and builds a federation consisting of a zone for each edge. In order to get the missing zones, this federation is subtracted from the full federation built in the beginning. The result of the subtraction is then used to build an edge for each zone and add it to the automaton's edges.

5.10 Duplicate automata in the refinement query

When running a refinement query, if the same automaton is present more than once on either side of the refinement, then problems can arise. Such a scenario can occur in a self refinement query, but also in more complex queries containing compositions or conjunctions. The main issue that we discovered is in the manip-

ulation of zones, as applying constraints and updates on a clock will do that on the first occurrence of said clock, which is not necessarily the correct occurrence, as the same automaton can appear multiple times.

One solution to this problem would be to make a deep copy of each automaton whenever it is seen, given that it is not its first occurrence. This way we can ensure that the set of clocks of an automaton and the set of clocks of its copy are not the same, so we have a reference to the clock of the right automaton.

Another solution is to display an error message, in the style of Ecdar 0.10, which reports a "Duplicate process instance" error and points to the automaton that appears more than once in the query.

We found both approaches reasonable and estimated that they would require roughly the same time to implement. In order to maintain a consistency between Jecdar 0.2 and Ecdar wherever it is sensible, we opted for reporting an error message in queries containing duplicate instances of an automaton.

5.11 Error logging

While Ecdar 2.x simply returns "yes", "no" or "maybe" as the result of a query, Ecdar 0.10 provides the user with significantly more feedback, namely it decides whether a query passes or fails, and in case of the latter it displays an error message pointing to the reasons why the chosen query did not succeed. Such messages include stating that there is a duplicate process instance, as presented in Section 5.10, or that an automaton is inconsistent.

We believe that offering the user the chance to see why a certain query has failed brings value and can contribute to the construction of correct models. For this reason, error messages were added to Jecdar 0.2.

This modification requires the different queries to check for all preconditions instead of terminating when the first issue is encountered. If any of the preconditions fails, then the refinement check is not run and the query fails, as expected. Having to check all preconditions can decrease performance in situations where the first issue is discovered early in the process, however it provides feedback and we think this trade-off is justified in the long run, especially as the size of models increases and they become harder to debug.

Figure 5.10 shows an example of the error messages that Jecdar 0.2 returns when a query fails.

5.12 Passed list as HashMap

As already mentioned, during the development of Jecdar we specifically focus on correctness of verification, especially given all the discovered inconsistencies with the theory in Ecdar 0.10. However, we understand that performance is a critical

```
-rq -xml ./samples/xml/ImplTests.xml refinement:G17<=G17 implementation:G14
false
Duplicate process instance: G17.
false
Automaton G14 is non-deterministic.
Automaton G14 is not output urgent.
```

Figure 5.10: Error logging example for multiple queries.

factor due to the existence of the state-space explosion problem that addresses the exponential growth of the state-space needed to be explored as the size of models increases. For that reason we outline performance optimization as one of the most relevant future improvements.

Nonetheless, in the implementation of Jecdar 0.2 we try to optimize performance at a micro-level. This mostly concerns the usage of data structures that provide better execution times. With the growing amount of test cases this problem becomes easier to address as the run time noticeably increases.

One of such performance improvements was done to the *passed* list of state pairs of the refinement relation. Previously, it was stored as a list of state pairs, which was a reasonable choice given the unpredictable final size of the list and having to constantly add elements to it. However, as with larger models the *passed* list increases, it becomes more and more expensive to check if a newly discovered state pair already belongs to that list.

To address this issue, we have implemented the *passed* list as a hash map, with the key being the locations of both left and right side of the refinement and the values being the list of relevant state pairs. Note that this slightly decreases performance for cases with a small *passed* list due to having to keep a key object, however it drastically increases performance for lists that are of medium and large size.

This change helped us to decrease the overall execution time of the entire test suite by around 10-20% for machines using Windows OS. On the other hand, for machines running MacOS, the performance has improved by up to 40% in terms of time.

5.13 Non-canonical refinement relation

After having finished implementing *determinism, consistency* and *implementation* checks, as well as global zones, described in Section 4.6, we achieved verification of the refinement feature that is correct and consistent with the theory. Hence, we proceed to implementing the functionality to return the refinement relation, which shows how the state-space was explored.

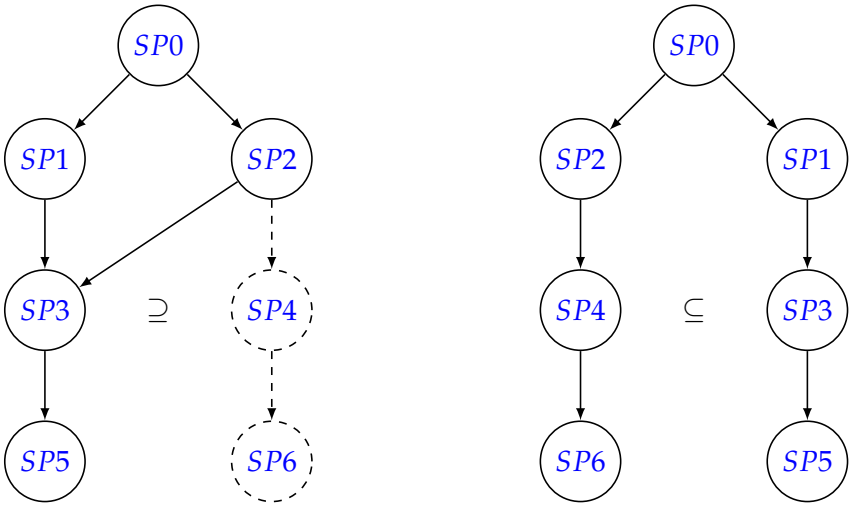


Figure 5.11: Two valid non-canonical refinement relations for the same automata participating in the refinement

5.13.1 Storing the refinement relation

The first challenge in returning the refinement relation became to correctly store it while preserving the order in which state pairs have been explored as well as the information about all the edges that are taken to transition from one state pair to another. Even though the *passed* list generated during the verification of the refinement feature contains all the explored state pairs, there is no information about the order in which these state pairs were explored or the edges taken. Therefore, for returning the refinement relation we chose to generate a graph along the verification of the refinement, where a node contains information about the corresponding state pair and transitions connecting these nodes keep track of all the edges of automata taken to reach a node.

Due to the fact that the directed graph that contains the refinement relation is being generated together with the *passed* list, the state space shown in the graph is not the complete one. As soon as a new state pair is discovered during the verification of refinement it is only stored and further explored if its zone is not a subset of any state pair that has previously been explored. This means that the size of the *passed* list and, as a result, the size of the refinement relation graph for the same automata might differ depending on the exploration order. Figure 5.11 shows a small such example of two possible refinement relation graphs for the exact same case of automata participating in the refinement. Note that in both graphs state pair **SP4** is in relation with state pair **SP3**, such that **SP4** $\subseteq$ **SP3**. Therefore, what determines the size of the graph is the order in which **SP3** and **SP4** are explored.

The graph on the left is the result of **SP3** being explored and stored in the

passed list first. All the state pairs derived from **SP3** will eventually be stored as well. However, **SP4** is only discovered, but never stored as it is a subset of **SP3**. In such a case we create an edge from **SP2** to **SP3** which also contains information about it leading to a superset state pair (**SP3**) instead of the original state pair **SP4**.

The graph on the right side showcases **SP4** being explored before **SP3**. Even though **SP4** $\subseteq$ **SP3**, it gets stored since **SP3** is not present in the *passed* list yet. As a result, the graph on the right ends up storing two more nodes than the graph on the left side, solely because of the state space exploration order. Therefore, such a refinement relation is considered non-canonical.

5.13.2 Printing the refinement relation

Once the refinement relation is computed, we must find a way to print it out. There are two possibilities for printing the relation; either in a nested JSON format or as two separate lists, one containing state pairs and the other containing the transitions between them. Both of these ways are valid to be implemented as both of them are meant to be parsed and used in the GUI. We have chosen to print the refinement relation in a JSON format. An example of the JSON format refinement relation for the query `refinement:T6<=T5` of two automata from Figures 3.8 and 3.9 can be seen in Appendix A.

Currently, the refinement relation that is returned contains information about locations, zones and edges. Ultimately, it is meant to be used in a GUI to highlight the edges of automata that were taken and the locations that were reached. However, to do so an additional logic would have to be added to the GUI to determine which edge is taken. To avoid that, one could assign IDs to edges in the GUI, which would then be stored in models. In its turn, Jecdar 0.2 would return IDs of edges which could be used by the GUI to quickly determine edges that require highlighting. Unfortunately, neither Ecdar 0.10 nor Ecdar 2.x assigns unique identifiers to edges in their current implementation. Consequently, it is unreasonable to provide this information without modifying the GUI, in order to allow it to create and store IDs on the edges. For the time being, transitions consist only of references to the source and target state pair, together with the corresponding channel.

5.14 Jecdar commands

As mentioned in Kiviriga et al. (2019), we have several commands that can be used for communicating with Jecdar from the terminal. The list of commands can be seen in Table 5.1. The "-rq folderPath query query..." command, which runs one or more queries, given the path of the model, has been modified with the addition of an extra flag to indicate whether the model is in JSON or in XML format. The reason for doing this is so that the controller can decide which parser it should use.

In order to run such a command, "-rq" must be typed, followed by either "-json" or "-xml", followed by the actual queries.

Moreover, JECDAR 0.2 is now capable of recognizing such keywords as determinism, consistency and implementation, which correspond to the newly implemented features. These keywords are parts of the query and can be invoked as follows:

- **determinism:** `-rq -xml path determinism:Automaton`.
  In ECDAR 0.10 and ECDAR 2.x the determinism check as a stand alone check does not exist, however it is included in the consistency and implementation checks.

- **consistency:** `-rq -xml path consistency:Automaton`.
  The consistency check first performs a determinism check and then verifies the independent progress property. Note that this command will only perform the least fixpoint consistency check. The full consistency check is only used as part of the implementation check and cannot be explicitly used as a stand alone command.

- **implementation:** `-rq -xml path implementation:Automaton`.
  The implementation check first performs a determinism check, then the full consistency check and lastly it checks if the output urgency property holds for the entire automaton.

In addition to the previously mentioned commands, we introduced one more new command in order to be able to return the refinement relation. Table 5.1 provides the description of the command, as well as an example of how to run the command. The flat indicating whether the model is in JSON or XML format must be provided. The "-rqrrr" command returns the refinement relation in a tree format as well as the result of the check. Just like the "-rq" command, "-rqrrr" is capable of dealing with multiple queries at the same time and returning the relation.

Commands	Description of commands
-help	This command provides a list of all JECDAR commands as well as their usage
-version	Returns the current version of the engine.
-vq query	Given a query this command will verify if the query is syntactically correct.
-rq -json/-xml folderPath query query...	Given a folder location with the flag for the format of the files and one or more queries this command will firstly check if the queries are syntactically correct and then will execute them.
-rqrrr -json/-xml folderPath query query...	Given a folder location with the flag for the format and one or more queries the command will execute the refinement check and return the refinement relation.

Table 5.1: List of current JECDAR 0.2 commands

Chapter 6

Testing

In this chapter we describe the testing approach as well as some specific examples of testing scenarios that help us to validate correctness and quality. The testing experience from the previous semester of JECDAR development, which included over 130 tests with an overall code coverage being at least 95 percent, has proven to be insufficient to detect some of the refinement problems. Therefore, we have introduced a number of case specific tests with the aim of testing every potential corner case of both previously existing and newly implemented features.

These test cases include all of the examples described in Chapters 3 and 4, as well as a number of their variations with potentially differing values, equality signs, amounts of clocks and so on. Moreover, the complete suite of tests also includes the refinement check for most of the automata where each of them is challenged to refine itself. This helps to eliminate the possibility of JECDAR 0.2 having incorrect "self" refinement semantics, as in the case of ECDAR 0.10 (described in Section 3.1).

This chapter includes a description of determinism, consistency and implementation testing details as well as several interesting test cases with the corresponding explanation of those.

6.1 Determinism tests

One of the most important tests in the determinism check is of the automaton illustrated in Figure 6.1. By using such an automaton in a test case we ensure that JECDAR 0.2 does not make the same mistake as ECDAR 0.10 - no inclusion of multiple edges of the same action from the source location to the same target location in further determinism check (as described in Section 3.8). Even though both edges from **id0** to **id2** can be overlapping due to the fact that they lead to the same location, each of them individually must participate in further determinism check with the edge from **id0** to **id1**. In JECDAR 0.2 this results in not satisfied check

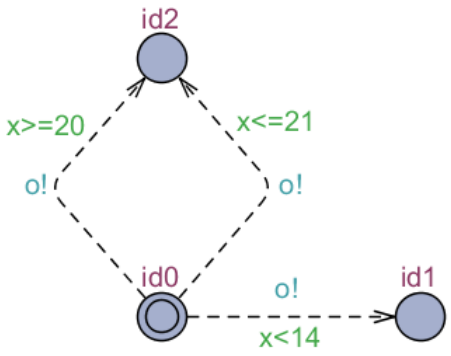


Figure 6.1: Non-deterministic automaton test case, which is not caught by ECDAR 0.10.

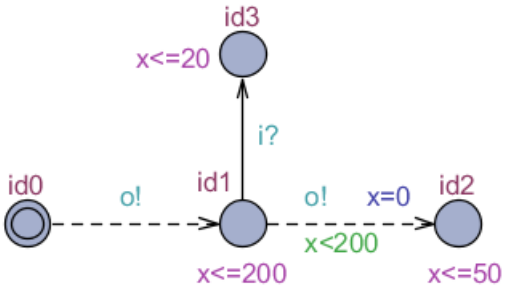


Figure 6.2: Automaton that will only satisfy the least fixpoint consistency check.

due to the presence of a non-deterministic choice for two edges leading to different locations: the edge from **id0** to **id1** with the $x < 14$ guard and the one from **id0** to **id2** with the $x \leq 21$ guard. Note that the same test case gives a false positive result in ECDAR 0.10.

6.2 Consistency tests

An important part of the tests that were performed are the consistency tests. We have tried to create automata which would include the rarest corner cases. The case illustrated in Figure 6.2, where the consistency check starts in location **id0** satisfies the least fixpoint consistency check, but not the greatest fixpoint consistency check. It is important to remember that the least fixpoint consistency permits pruning of output edges if the property of independent progress has already been satisfied for a given location. Since the property of independent progress is satisfied already in location **id0** and there are no outgoing input edges, which are mandatory to explore, the algorithm prunes the rest of the automaton.

Note how the rest of the automaton does not satisfy the property of independent progress. In location **id1** the independent progress property is violated by the

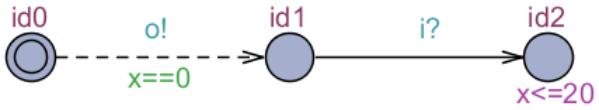


Figure 6.3: Automaton that fails full consistency check that is required by implementation check.

corresponding invariant ($x \leq 200$) which disallows delaying indefinitely. Moreover, both outgoing edges from location **id1** must then be explored. Since the targets of those edges cannot ensure independent progress either, we conclude that none of the three mentioned locations would satisfy the consistency check. With that being said, this test case automaton can only satisfy the least fixpoint consistency check due to the pruning of the rest of the automaton, except for location **id0**.

6.3 Implementation tests

The defining property of the implementation check is the one of the output urgency. It requires any output edge to be traversable only with a single clock valuation, without the possibility to delay instead. Consider the test case illustrated in Figure 6.3. This automaton is consistent and the only output edge it has is urgent, which satisfies the property of the output urgency for the entire automaton.

Moreover, it is important to remember that the implementation check also requires a consistency check that is different from the least fixpoint consistency check mentioned previously in this chapter. Since the property of output urgency concerns all output edges, the corresponding consistency check should not be allowed to prune these edges, resulting in having to perform a full consistency check, which was described in Section 3.7. Thus, having an invariant on location with **id2** makes the implementation check fail due to the property of independent progress not being satisfied.

6.4 Summary

To conclude the chapter, more thorough and detailed testing was done for all the existing and newly implemented features. In particular, we focused on testing all the corner cases that were discovered during this project as well as cases where Ecdar 0.10 has shown to be inconsistent with the theory.

In total there are currently more than 330 quality tests in Jecdar 0.2, which is 200 more than in the previous version of the engine. The code coverage of the most vital functionality related to feature verification has also been improved from previously achieved more than 95% to a complete 100% coverage, as can be seen

Element	Class, %	Method, %	Line, %	Branch, %
Composition	100% (1/1)	100% (11/11)	100% (68/68)	100% (20/20)
Conjunction	100% (1/1)	100% (8/8)	100% (31/31)	100% (8/8)
Refinement	100% (1/1)	100% (13/13)	100% (135/135)	97% (38/39)
SimpleTransitionSystem	100% (1/1)	100% (16/16)	100% (121/121)	95% (39/41)
TransitionSystem	100% (1/1)	100% (15/15)	100% (80/80)	100% (17/17)

Figure 6.4: Code and branch coverage of the most important verification algorithms.

in Figure 6.4. What is more important, in its implementation JECDAR 0.2 does not contain the same inconsistencies with the theory as discovered in ECDAR 0.10.

In addition to the line coverage metrics, we have also considered branch coverage. It measures if all decision outcomes have been tested, where a decision can be *if* and *case* statements, *while* and *for* loops. This metrics shows if all branches (possible cases of decisions) were ever reached and consequently if the test case suite covers all possible decisions.

The current test case suite of JECDAR 0.2 achieves a minimum of 95% branch coverage (Figure 6.4). The missing percentage of the branch coverage for *Refinement* and *SimpleTransitionSystem* classes is related to checking validity of the DBM zone, which appears to never be invalid and thus does not explore the negative outcome of an IF statement. However, such cases do not influence the correctness of the algorithms as they are validity checks that appear to never fail. Therefore we deem them not important.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The result of our work spanning two semesters, Jecdar 0.2, is an engine for modelling and verifying properties of real-time systems based on the specification theory of timed input/output automata defined in David et al. (2010b). Although there already exists an integrated environment that can do this, namely Ecdar, this tool requires a license for non-academic use. Our engine addresses this issue by being open-source and it is meant to be integrated with a GUI that allows efficient creation of models and verification of properties.

Jecdar 0.2 is not a finished tool, as it lacks some of the features available in Ecdar, such as computing quotient, running models in a simulator and checking TCTL formulas. The aim of the project was to implement features that perform correctly, which meant that the goal of implementing as many features as possible was downgraded. Currently, Jecdar 0.2 can be used to check features such as consistency, implementation and refinement and it can operate on the composition and conjunction of multiple transition systems.

The result of this process is not an engine that behaves exactly like Ecdar, meaning that there are several implementation differences between the two engines. A large part of our process consisted of discovering cases where Ecdar displays inconsistencies with the theory and devising ways to tackle said cases.

This report documented our initial goal for the project, observations regarding scenarios that display inconsistencies between Ecdar and the theory of timed input/output automata, concepts that we used for the purpose of correctly implementing certain features (both successful and unsuccessful), implementation details and a description of our testing approach.

7.2 Future Work

In this chapter we introduce some prominent ideas for future work that can be conducted on the Jecdar 0.2 engine. Note that this chapter does not include the points from the future work described in our previous semester, which can be found in Kiviriga et al. (2019).

7.2.1 Greatest fixpoint consistency

Currently, Jecdar 0.2 includes the least fixpoint consistency check as well as the full consistency check used in the implementation check. An interesting subject for future work can however be the computation of the greatest fixpoint consistency, described in Section 3.7.

7.2.2 Support user-defined variables

Ecdar 0.10 supports record variables, which are specified by the keyword *struct*, which follows the *C* notation. An example of a record *k* can be seen in Listing 7.1. Thus, user-defined variables can be a good candidate for new features in Jecdar 0.2.

```
struct{
    int x;
    int y;
} k;
```

Listing 7.1: Record example

7.2.3 TCTL formulas

Since Ecdar 0.10 is built on top of Uppaal it includes TCTL formulas, which could be a subject for improvement in Jecdar.

7.2.4 Returning trace

A relevant addition to the refinement check is being able to return the trace that leads to an error state in cases where the refinement does not hold. This feature would help the user debug incorrect models by pinpointing the sequence of transitions and the time constraints that cause the system to reach an error state.

A trace can also be regarded as a subset of the TCTL formulas, since internally it would behave as a reachability check.

The computation of the trace would have to operate on the refinement relation that we represent as a graph, as described in 5.13. It is worth mentioning that an

error state can be reached from more than one trace, as there could be several paths leading to that state.

A potential challenge that we believe would be faced is back-propagating the information about the zone contained in the error state in order to adjust the zones of the states that lead to said state.

Moreover, one may decide to implement the possibility to return all possible traces to the given state. This would require the generation of the complete refinement relation.

7.2.5 State-space reduction techniques

The field of model-checking is known for having the problem of exponential state-space growth during its exploration. This can be another improvement that JECDAR 0.2 can benefit from - the introduction of state-space reduction techniques. Some of such techniques can greatly reduce the amount of time and memory required during state-space exploration.

As already mentioned previously our focus is on correctness rather than performance. At the moment, JECDAR 0.2 is making use of a rather primitive implementation of extrapolation abstraction. Hence, a very promising potential performance improvement could include the implementation of more advanced extrapolation abstraction techniques, such as location-based bounds for extrapolation or lower and upper bound dependent extrapolation. For some models with very big values of the time-progress conditions (guards and invariants) this could greatly reduce the size of the state-space that has to be explored.

Another case of abstraction could be symmetry reduction. It is based on the *state swaps* that are used to reduce the size of the state-space that has to be explored, consequently reducing the time and memory needed for verification. According to Hendriks et al. (2004), to achieve that a suitable bisimulation equivalence has to be derived from the system description, introducing the necessity for the engine to support such declarations. Afterwards, the representative function is constructed and used to convert a state into a corresponding equivalence class. Then when doing a reachability analysis it suffices to explore and store only one representative of the equivalence class.

Bibliography

[1] Gerd Behrmann et al. "Lower and upper bounds in zone-based abstractions of timed automata". In: *International Journal on Software Tools for Technology Transfer* 8.3 (2006), pp. 204–215. ISSN: 1433-2787. DOI: 10.1007/s10009-005-0190-0. URL: https://doi.org/10.1007/s10009-005-0190-0.

[2] Gerd Behrmann et al. "Static Guard Analysis in Timed Automata Verification". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Hubert Garavel and John Hatcliff. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 254–270. ISBN: 978-3-540-36577-8.

[3] Gerd Behrmann et al. "UppaaL Implementation Secrets". In: *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Ed. by Werner Damm and Ernst Rüdiger Olderog. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 3–22. ISBN: 978-3-540-45739-8.

[4] Alexandre David et al. "ECDAR: An Environment for Compositional Design and Analysis of Real Time Systems". In: *Automated Technology for Verification and Analysis*. Ed. by Ahmed Bouajjani and Wei-Ngan Chin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 365–370. ISBN: 978-3-642-15643-4.

[5] Alexandre David et al. "Timed I/O Automata: A Complete Specification Theory for Real-time Systems". In: *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control*. HSCC '10. Stockholm, Sweden: ACM, 2010, pp. 91–100. ISBN: 978-1-60558-955-8. DOI: 10.1145/1755952.1755967. URL: http://doi.acm.org/10.1145/1755952.1755967.

[6] Alexandre David et al. "Uppaal Stratego". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Christel Baier and Cesare Tinelli. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 206–211. ISBN: 978-3-662-46681-0.

[7] *DBM manual*. Apr. 16, 2019. URL: http://people.cs.aau.dk/~adavid/UDBM/manual-061023.pdf.

[8] Martijn Hendriks et al. "Adding Symmetry Reduction to Uppaal". In: *Formal Modeling and Analysis of Timed Systems*. Ed. by Kim Guldstrand Larsen and Peter Niebert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 46–59. ISBN: 978-3-540-40903-8.

[9] Andrej Kiviriga, Napalys Klicius, and Cristina Simionescu. "JECDAR: A Java Engine for Compositional Design and Analysis of Real Time Systems". In: Aalborg, Denmark, 2019.

[10] Kim G. Larsen, Paul Pettersson, and Wang Yi. "Uppaal in a nutshell". In: *International Journal on Software Tools for Technology Transfer* 1.1 (Dec. 1997), pp. 134–152. ISSN: 1433-2779. DOI: 10.1007/s100090050010. URL: https://doi.org/10.1007/s100090050010.

[11] Stavros Tripakis and Costas Courcoubetis. "Extending promela and spin for real time". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Tiziana Margaria and Bernhard Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 329–348. ISBN: 978-3-540-49874-2.

[12] *Uppaal DBM Library*. Apr. 16, 2019. URL: http://people.cs.aau.dk/~adavid/UDBM/index.html.

Appendix A

Refinement relation in JSON format

```
{
"left": "id45",
"zone": "[1, 1, 1, 2147483646, 1, 1, 2147483646, 1, 1]",
"right": "id41",
"initial sp id": "0",
"transitions": [{
    "source sp id": "0",
    "channel": "ro",
    "target sp id": "1",
    "target sp": {
      "left": "id44",
      "zone": "[1, 1, 1, 13, 1, 1, 13, 1, 1]",
      "state pair id": "1",
      "right": "id40",
      "transitions": [{
          "source sp id": "1",
          "channel": "go",
          "target sp id": "2",
          "target sp": {
            "left": "id42",
            "zone": "[1, 1, 1, 13, 1, 1, 13, 1, 1]",
            "state pair id": "2",
            "right": "id40",
            "transitions": [{
                "source sp id": "2",
                "channel": "ri",
```

```
                "target sp id": "4",
                "target sp": {
                  "left": "id43",
                  "zone": "[1, 1, 1, 2147483646, 1, 1, 2147
                     483646, 1, 1]",
                  "state pair id": "4",
                  "right": "id39"
                }
              }
            ]
          }
        }, {
          "source sp id": "1",
          "channel": "ri",
          "target sp id": "3",
          "target sp": {
            "left": "id44",
            "zone": "[1, 1, 1, 13, 1, 1, 13, 1, 1]",
            "state pair id": "3",
            "right": "id39",
            "transitions": [{
                "source sp id": "3",
                "channel": "go",
                "target sp id": "5",
                "target sp": {
                  "left": "id42",
                  "zone": "[1, 1, 1, 13, 1, 1, 13, 1, 1]",
                  "state pair id": "5",
                  "right": "id39",
                  "transitions": [{
                      "source sp id": "5",
                      "channel": "ri",
                      "target sp id": "4"
                    }
                  ]
                }
              }
            ]
          }
        }
      ]}}]}
```