TNO innovation for life

ESI
Powered by industry, academia and TNO

Inferring an analysis model from execution logs

From PPS to LSAT

TNO Public ⟩ TNO 2023 R12452
15 December 2023

ICT, Strategy & Policy
www.tno.nl
+31 88 866 50 00
info@tno.nl

TNO 2023 R12452 – 15 December 2023

From PPS to LSAT

Inferring an analysis model from execution logs

Author(s)	Jacques Verriet

Classification report	TNO Public
Title	TNO Public
Report text	TNO Public
Number of pages	24 (excl. front and back cover)
Number of appendices	0
Sponsor	ASML
Programme name	Maestro
Project name	Maestro 2023
Project number	060.55514/01.01

Our partners

The research is carried out as part of the Maestro program under the responsibility of TNO-ESI with ASML as the carrying industrial partner. The Maestro research is supported by the Netherlands Organisation for Applied Scientific Research TNO.

Contents

1 Introduction

In a model-based development way of working, models are used to specify a system and to automatically analyse the specified system. By iteratively adapting the specification and analysing the specification's performance, one can explore the system's design space to come to a specification that is to be realised.

These specification models are often created manually, and this may involve a large effort. Manual modelling is natural for greenfield development as no system exists yet. In case of brownfield development, the model-based design space exploration typically starts from an existing system. To reduce the model creation effort, one could use information from the existing system to create a (partial) initial model.

This report addresses the (partial) creation of analysis models from execution logs. We illustrate this transformation using PPS [1] [2] and LSAT [2] [3]. PPS (Platform Performance Suite) is a tool to visualise and analyse timed message sequence charts (TMSC) [4], which capture a system's (timing) behaviour in terms of events and their dependencies. LSAT (Logistics Specification and Analysis Tool) is a tool to specify and analyse the timing behaviour of flexible manufacturing systems.

1.1 Outline

There are conceptual differences between PPS models and LSAT models. Chapter 2 explains these differences and describes a way to make PPS' TMSC models more suitable for translation into LSAT model. Chapter 3 illustrates the transformation of a TMSC into an LSAT model. The focus on Chapter 3 on the parts of the transformation that can and cannot be automated. Chapter 4 reflects on the illustrated approach.

2 TMSC pre-processing

Creating an LSAT model from a PPS model involves transforming a PPS TMSC into the four elements of an LSAT model [2]. As PPS is primarily meant for software systems with a CORBA-like architecture and LSAT is primarily targeted at mechanical systems, this transformation is not straightforward. In this chapter, we discuss a way to pre-process TMSCs such that they can more easily be transformed to an LSAT model. In Section 2.1, we discuss the different characteristics of PPS' TMSCs and LSAT models that make the transformation between them difficult. In Section 2.2, we describe one approach to reduce the distance between TMSCs and LSAT models.

2.1 PPS vs. LSAT

The input for the transformation from PPS to LSAT models is a Timed Message Sequence Chart (TMSC) [4], which involves a set of events and a set of timing constraint relations between these events. A simple example of an artificial TMSC is shown in Figure 2.1.

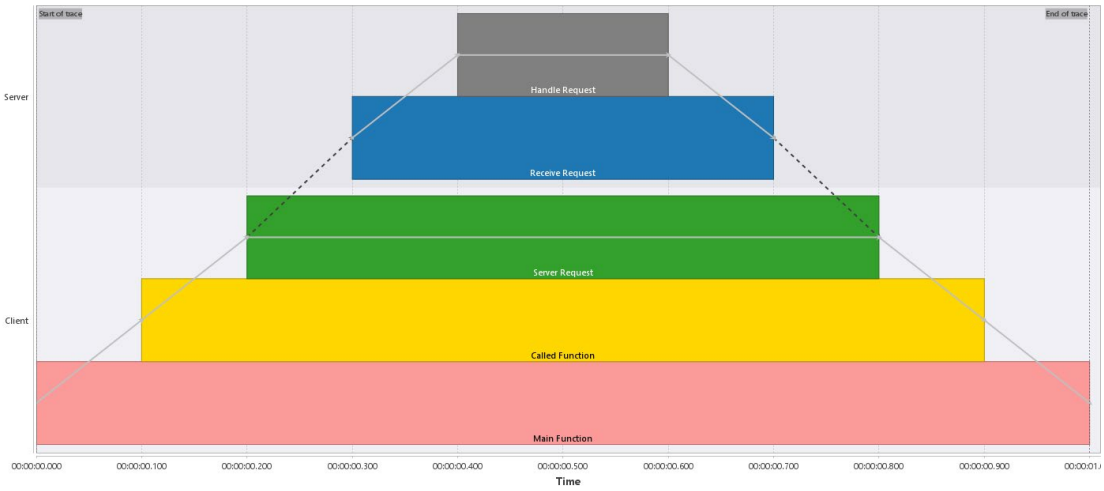

Figure 2.1: Input TMSC

Figure 2.1 shows the software call stacks of two software threads and the dependencies between the corresponding events. A TMSC may contain different types of dependencies:

- There are dependencies between the start and the end of one function. Two of these are shown in Figure 2.1. Similarly, there may be dependencies between the start and the end of a call stack.
- In case of a (remote) function call, there are dependencies between the start of the calling function and the start of the called function and between the end of the called function and the end of the calling function. These dependencies represent data being exchanged between the functions. The dashed dependencies in Figure 2.1 are the dependencies of a remote function call.
- In case of sequence dependencies, there are dependencies between the end of a predecessor function and the start of a successor function. Figure 2.1 does not include this type of dependencies.

With respect to dependencies between events, LSAT is much more restrictive than PPS: LSAT's activity models only allow dependencies between the end of an action/function and the start of another. Another difference between PPS' TMSC and LSAT's activity models is the fact that LSAT activity model dependencies do not have a duration: all communication is assumed to be instantaneous.

A third difference between PPS' TMSCs and LSAT models is the call stacks. PPS has been developed to visualise and analyse the timing behaviour of software systems and LSAT has been developed for the analysis of the timing behaviour of the hardware actions of flexible manufacturing systems.

2.2 Collapsing TMSC call stacks

To facilitate a transformation from PPS' TMSC to LSAT models, we should address the differences identified in Section 2.1. We use the TMSC in Figure 2.1 as an illustration on how one could deal with the differences.

To eliminate a TMSC's call stack structure, the call stack can be flattened. Figure 2.2 shows a TMSC in which the call stack of Figure 2.1 has been collapsed to the bottom functions of the stacks. To capture the dependencies of the original call stack structure, these bottom functions are split at the synchronisation points of the original TMSC. Note that the start and end times of the call stacks are identical in Figure 2.1 and Figure 2.2. Also the timing of the dependencies (between the call stack) is identical.

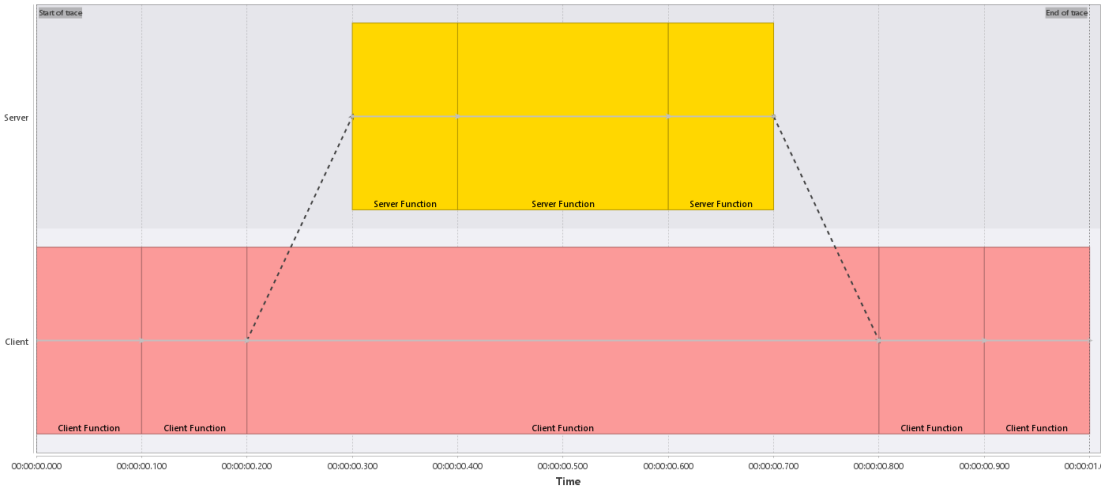

Figure 2.2: Flattened TMSC

The dependencies in the TMSCs in Figure 2.1 and Figure 2.2 are not instantaneous; in fact, there are significant communication delays. To have these delays appear in an LSAT model, the communication delay can be modelled as a function. Figure 2.3 shows a TMSC which augments the TMSC in Figure 2.2: a network swim lane has been introduced. The TMSC in Figure 2.3 replaces the dependencies in Figure 2.2 by two instantaneous dependencies. To capture the duration of the original dependencies, network communication functions are introduced.

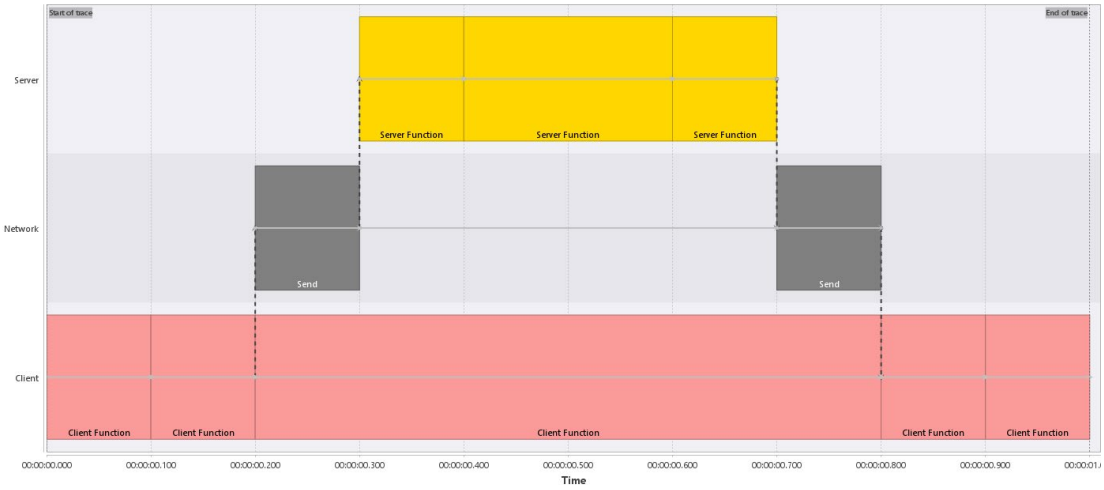

Figure 2.3: Flattened TMSC with network functions

After these changes, a TMSC has been transformed into a TMSC resembling an LSAT's activity model.

2.3 Summary

In this chapter, we have presented the main differences between PPS' TMSCs and LSAT model. In addition, we have presented an automatable TMSC transformation, which makes a TMSC conceptually similar to LSAT activity models.

3 Transformation example

In this chapter, we explain a method to create an LSAT analysis model from a timed message sequence chart in PPS. We use a timed message sequence chart (TMSC) from Jonk [4] as a starting point. This model is explained in Section 3.1. Section 3.2 explains how the input TMSC is transformed into a TMSC which is suited for translation into an LSAT model. The transformation into an LSAT model is explained in Section 3.3. Section 3.5 illustrates the type of analysis that can be done once an LSAT model has been created. Section 3.6 summarises the transformation and reflects on the level of automation that can be achieved.

3.1 Input TMSC model

We use a TMSC model from Jonk [4] as an example to create an LSAT model. This TMSC is shown in Figure 3.1. This TMSC was manually created in PPS using Figure 8 of Jonk [4] as a reference. This TMSC captures the wafer exposure loop of a lithography scanner. For each die on a wafer, two cyber actions are performed: (pink) *Compute* actions and (yellow) *Queue* actions. The Compute action computes the setpoints to be tracked for the wafer exposure control loop. The consecutive Queue action queues these setpoints in a buffer.

After these cyber steps, the actual exposure is performed. This exposure involves two physical actions: (grey) *Step* actions and (blue) *Scan* actions. The Step action positions the wafer for the exposure of the next die and the Scan action performs the die exposure. As these steps use the queued setpoints, there is a dependency between the cyber actions and the physical actions: there are arcs from the end of the Queue action to the start of the Step action.

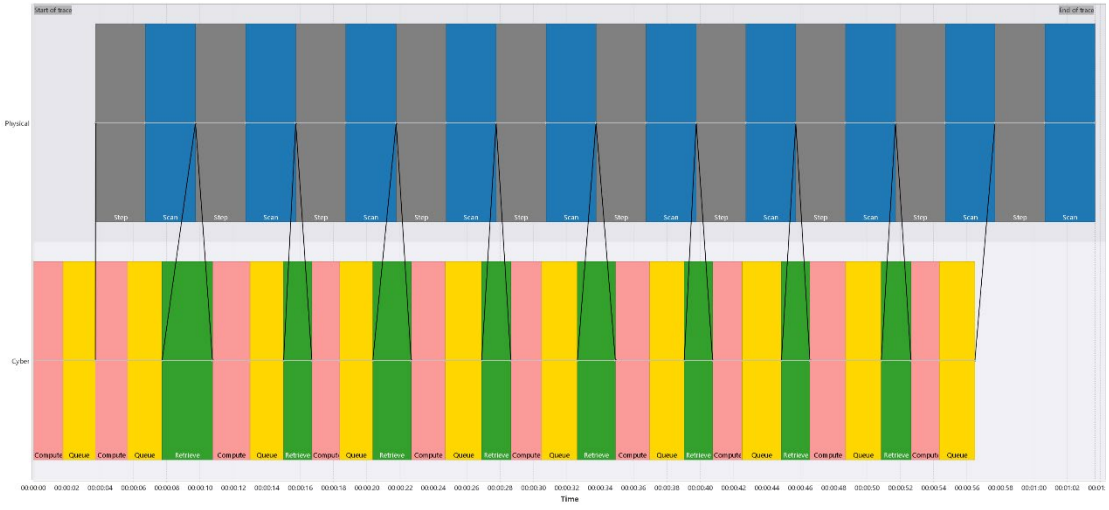

Figure 3.1: Input TMSC (recreated from Jonk [4])

To optimise exposure accuracy, information about physical disturbances encountered during Step and Scan actions is used in the Compute step. This dependency is realised by the (green) *Retrieve* actions, which retrieves information from the (running) Step and Scan

actions. The retrieved information is used for the Compute and Queue actions of the next die to be exposed. This is captured by the dependencies between the end of Scan actions and the end of Retrieve actions, which coincide with the start of Compute actions.

Note that there is no Retrieve action between the cyber actions of the first and second die: the retrieval of information from the exposure of the first die is input for the Compute and Queue actions for the third die. This corresponds to a buffer size of two [4]. With a buffer size of one, the exposure of the second die would be delayed, leading to a lower exposure throughput. On the other hand, the exposure would use more recent disturbance information, which could lead to higher-quality exposure.

3.2 TMSC pre-processing

The TMSC model discussed in Section 3.1 does not match the dependencies of LSAT [2] [3]. Retrieve actions run in parallel to the Step and Scan actions from which they gather information. This requires the Retrieve action to end after the Scan action. This is something that LSAT cannot capture in its activity language.

This issue can be resolved by adapting the TMSC using domain knowledge. There are several ways to do this. Deciding which approach to take requires domain knowledge, e.g. regarding which actions are most important:

- *Action removal*: The simplest manner would be removing all Retrieve actions from the TMSC and replacing the dependencies between the end of a Scan action and the end of a Retrieve action by dependencies between the end of a Scan action and the start of a Compute action. The resulting TMSC is shown in Figure 3.2.
- *Action splitting*: A second way to resolve the issue is similar to the approach presented in Chapter 2: the Retrieve action can be split into a Wait action and a Retrieve action. The dependencies between the end of the Scan action and the end of the Retrieve action are replaced by dependencies between the end of the Scan action and the start of the new Retrieve actions. The resulting TMSC is shown in Figure 3.3: the (green) Retrieve actions in Figure 3.1 have been replaced by a (brown) Wait action, which starts directly after the (yellow) Queue action and ends together with the (blue) Scan action, and a (green) Retrieve action.
- *Action shortening*: A third way is simpler than the second but is based on the same principle. Instead of splitting the Retrieve actions into two actions, the Retrieve actions is shortened. This is done by delaying the start of Retrieve actions until the end of Scan actions and introducing a dependency between the end of a Scan action and the start of the consecutive Retrieve action. The resulting TMSC is shown Figure 3.4; its (green) Retrieve actions now start directly after the (blue) Scan actions.

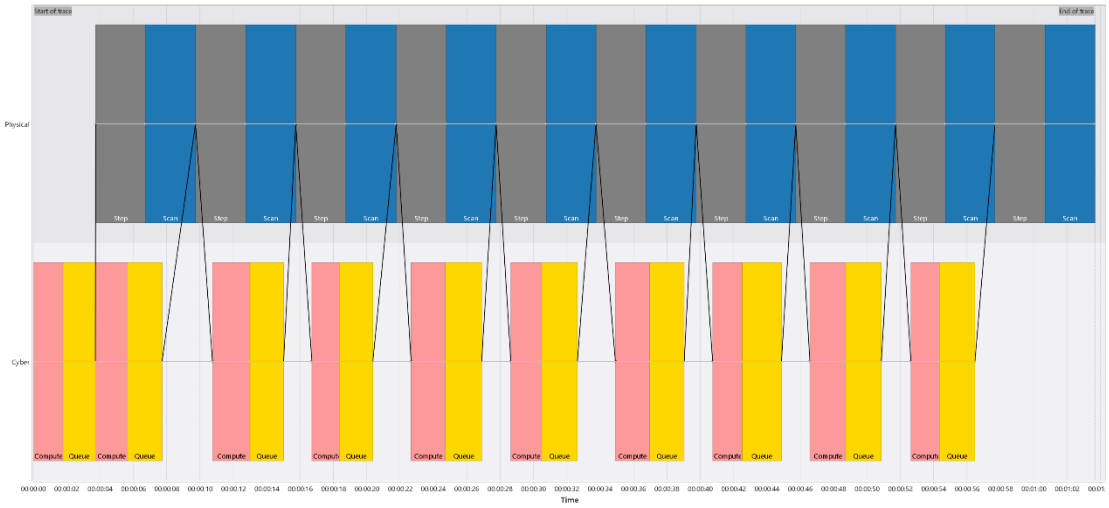

Figure 3.2: TMSC after removing the (green) Retrieve actions

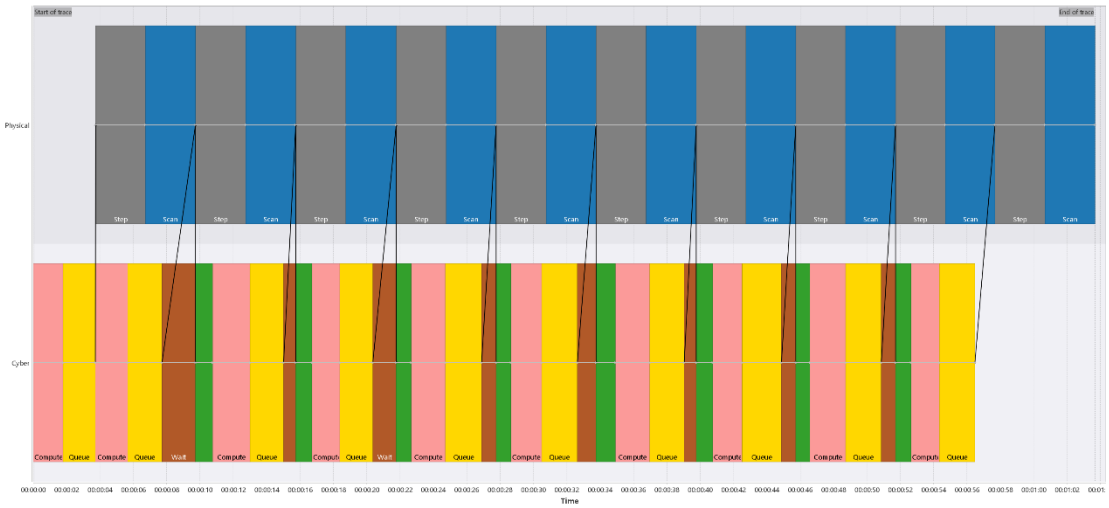

Figure 3.3: TMSC after splitting the (green) Retrieve actions in (brown) Wait and (green) Retrieve actions

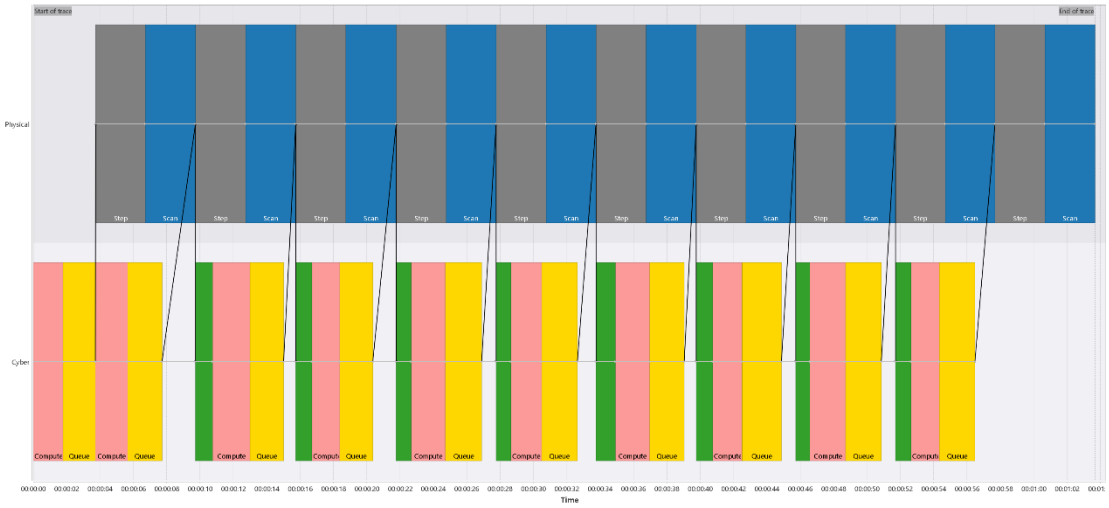

Figure 3.4: TMSC after delaying the start of (green) Retrieve actions

For this illustration, we have chosen the third way, i.e. the Retrieve actions are shortened by delaying their start time. The resulting TMSC is shown in Figure 3.4. We use this TMSC as the basis for creating an LSAT model.

Note that the TMSC pre-processing may change the timing of a TMSC. Whether this is acceptable is a decision to be made using application knowledge. The purpose of the application considered in this chapter is maximising the exposure throughput. As the selected pre-processing does not change the timing behaviour of the Step and Scan actions, shortening the Retrieve actions does not influence the exposure throughput.

3.3 LSAT model creation

An LSAT model consists of four elements: a machine specification, a setting specification, an activity specification and a dispatching specification. The creation of these specifications is explained in Sections 3.3.1, 3.3.2, 3.3.3 and 3.3.4. For each of these LSAT specifications, we address to what extent the specification could be generated fully automatically.

3.3.1 Machine specification

An LSAT machine specification defines the available resources and their peripherals. To create a machine specification, we create a peripheral type for each of the swim lanes of the input TMSC. The actions of these peripheral types correspond to the different actions in the TMSC's swim lane. In addition, we create a resource for each swim lane; this resource has a single peripheral which has the swim lane's peripheral type.

Figure 3.5 shows the machine specification corresponding to the TMSC in Figure 3.4. Most of this specification can be generated automatically using the rules explained above.

The PrepareResult and ExecuteResult resources are not shown in Figure 3.4. These resources are added for activity synchronisation, which is addressed in Sections 3.3.3 and 3.3.4. These sections explain how such synchronisation resources could be introduced automatically.

What cannot be generated automatically is the number of instances of these synchronisation resources. The machine specification in Figure 3.5 has two instances of

each synchronisation resource. This is needed to capture the 2-place buffer between preparation and exposure.

```
Machine PPS2LSAT

PeripheralType Physical {
  Actions {
    Step
    Scan
  }
}

PeripheralType Cyber {
  Actions {
    Compute
    Queue
    Retrieve
  }
}

Resource Physical {
  p: Physical
}

Resource Cyber {
  c: Cyber
}

Resource Buffer(1, 2) {
}

Resource PrepareResult(1, 2) {
}

Resource ExecuteResult(1, 2) {
}
```

Figure 3.5: Textual machine specification

Figure 3.6 shows the graphical machine specification, which corresponds to the textual specification in Figure 3.5.

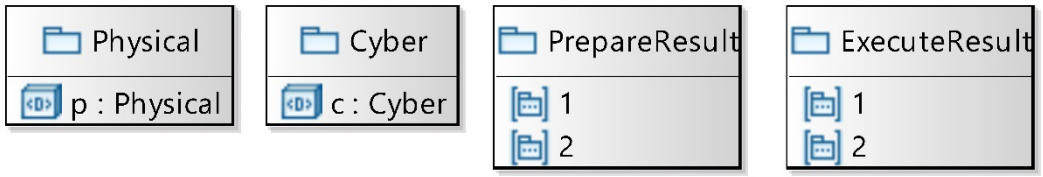

Figure 3.6: Graphical machine specification

3.3.2 Setting specification

An LSAT setting specification can fully be derived from a TMSC model. There are typically multiple instances of the same action in a TMSC swim lane. Each instance may have a unique duration. These durations are all used in the LSAT setting specification, which has the **Array** keyword to specify all durations of a peripheral action.

Figure 3.7 shows the setting specification corresponding to the (transformed) TMSC in Figure 3.4.

```
import "PPS2LSAT.machine"

Cyber.c {
  Timings {
    Compute = Array(1.7890, 1.9370, 2.290, 1.679, 2.056,
                    1.845, 2.058, 1.770, 2.145, 1.707)
    Queue = Array(1.934, 2.059, 1.975, 1.976, 2.172,
                  2.143, 2.069, 2.355, 2.136, 2.136, 2.126)
    Retrieve = Array(1.022, 0.967, 0.897, 0.884,
                     1.163, 0.984, 0.847, 0.928)
  }
}

Physical.p {
  Timings {
    Step = Array(2.992, 3.020, 3.001, 3.003, 3.004,
                 3.012, 2.994, 2.985, 2.993, 3.002)
    Scan = Array(2.998, 2.995, 3.022, 2.992, 2.986,
                 3.000, 2.984, 2.996, 2.975, 2.996)
  }
}
```

Figure 3.7: LSAT setting specification

The machine specification explained in Section 3.3.1 includes synchronisation resources that are not present in an input TMSC model. As these synchronisation resources do not have peripherals, they do not show up in a setting specification. This also means that an LSAT setting specification could be generated completely from a TMSC.

3.3.3 Activity specification

An LSAT activity specification specifies the behavioural building blocks of a system. An activity specification consists of multiple activities. An LSAT activity is represented by a directed acyclic graph containing claims and releases of resources, actions by peripherals and their synchronisations. The goal is to describe the recurring behaviour in an input TMSC using LSAT activities.

Creating an activity specification from a TMSC requires more domain knowledge than creating a machine specification. To partially generate an LSAT activity, one needs to specify which (recurring) actions in the TMSC form an activity. In the (pre-processed) TMSC in Figure 3.4, there are five recurring actions: Compute, Queue, Retrieve, Step and Scan. In this illustrative example, these are divided over two activities:

- The Compute and Queue actions are combined in a *Prepare* activity, and
- The Retrieve, Step and Scan actions are combined in an *Execute* activity.

We choose for these two activities as they are logically decoupled. The Prepare activity computes exposure setpoints and the Execute uses these setpoints. The synchronisation resource PrepareResult is used for the handover of the setpoints. Similarly, the Execute activity generates disturbance data, which is used by the Prepare activity. For this synchronisation, we use the ExecuteResult resource.

DEPENDENCIES

The dependencies between the actions in an activity can to a large extent be derived from the TMSC. If there is a dependency between the end of one action and the start of another action of the same activity, then the corresponding activity will “inherit” this dependency.

In an LSAT activity, the execution of an action must be preceded by a *claim* of the corresponding resource and followed by a *release* of this resource. These claims and releases and the corresponding dependencies can be generated automatically as well.

SYNCHRONISATION BARS

Peripheral actions, resource claims and resource releases in an LSAT activity may have at most one direct predecessor and at most one direct successor. In a TMSC, however, an action may have multiple direct predecessors and multiple direct successors. To capture such dependencies in LSAT, one needs to introduce *synchronisation bars*. For instance, if an action A has two direct successor actions B and C, then the corresponding activity would have a synchronisation bar S and dependencies from A to S, from S to B, and from S to C.[1]

SYNCHRONISATION RESOURCES

By splitting the recurring actions in a TMSC into activities, one cuts dependencies. By dividing the functions of the TMSC in Figure 3.4 into a Prepare activity and an Execute activity, the dependencies between the Queue and Step actions and those between Scan and Retrieve actions are cut.

LSAT has two means to “restore” these cut dependencies: *synchronisation resources* and *events* allow synchronisation between different activities. We have chosen to use synchronisation resources, because events put additional constraints on an LSAT model.[2] One could automatically generate a synchronisation resource for every cut dependency. Let us consider the cut dependency between the Queue and Step actions. In the Prepare activity, the corresponding synchronisation resource should be released after the Queue action has finished. Moreover, in the Execute activity, this synchronisation resource should be claimed before the Step action starts.

Figure 3.8 shows the textual specification of the Prepare activity; Figure 3.9 shows the corresponding graphical representation. Note that the PrepareResult resource is claimed in parallel to the Queue action, which writes the setpoints for the Step and Scan actions. The ExecuteResult resource is claimed while the Compute action is executed. This is needed as the Compute action uses the results of a previous Execute activity to compute the new setpoints.

[1] Unlike peripheral actions, resource claims and resource releases, synchronisation bars may have multiple direct predecessor and multiple direct successors in an LSAT activity.
[2] These constraints involve matching pairs of messages sent and received. For the running example, the constraints require undesired duplication of activities.

This activity can (probably) not be generated fully automatically. The claims and releases and actions that occur in the input TMSC and the corresponding dependencies could be generated automatically. As explained above, the synchronisation bars, between the actions and the resource claims and releases can be generated automatically as well. These are S0, S1 and S2 in Figure 3.8 and the black bars in Figure 3.9.

It may be too hard to automatically introduce the synchronisation resources, e.g. the PrepareResult and ExecuteResult resources, and their dependencies in an activity. Domain knowledge is required to decide when to claim and release a synchronisation resource.

Note that introducing dependencies with synchronisation resources could require additional synchronisation bars. As explained above, this could be resolved automatically.

```
activity Prepare {
  actions {
    ClaC: claim Cyber
    RelC: release Cyber

    ClaR: claim PrepareResult
    RelR: release PrepareResult

    ClaE: claim ExecuteResult
    RelE: release ExecuteResult

    Comp: Cyber.c.Compute
    Queu: Cyber.c.Queue
  }

  action flow {
    ClaC -> |S0
    ClaE -> |S0
    |S0 -> Comp
    Comp -> |S1
    ClaR -> |S1
    |S1 -> Queu
    |S1 -> RelE
    Queu -> |S2
    |S2 -> RelC
    |S2 -> RelR
}
```

Figure 3.8: Textual representation of the Prepare activity

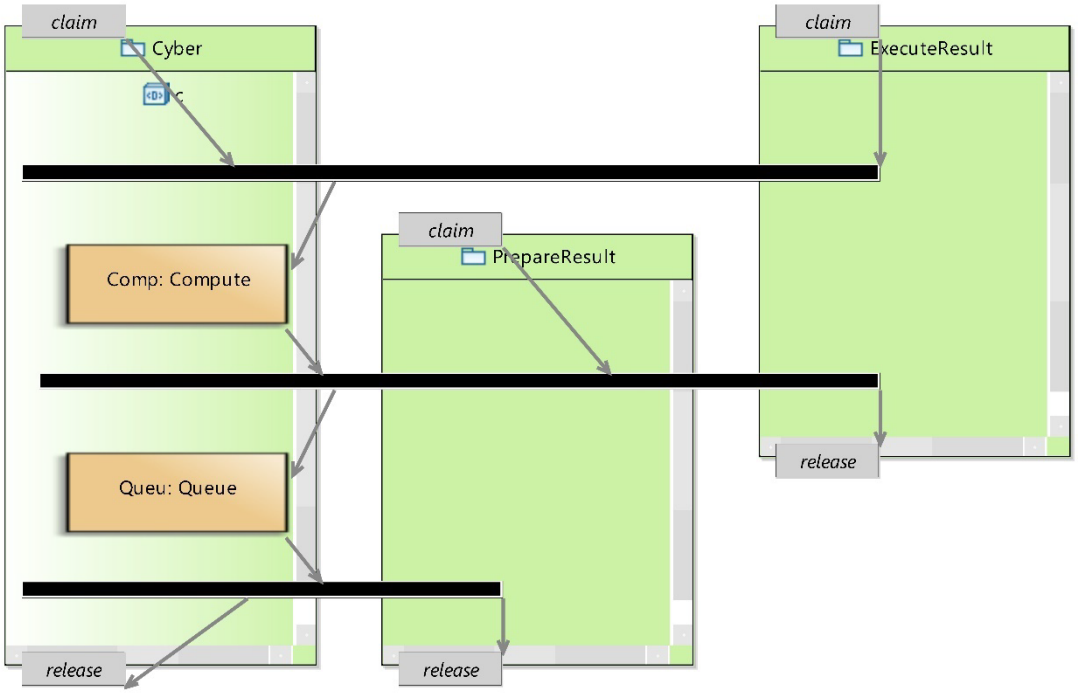

Figure 3.9: Graphical representation of the Prepare activity

Figure 3.10 and Figure 3.11 show the textual and graphical representations of the Execute activity. The PrepareResult synchronisation resource is used at the start of the activity. The claim of this resource corresponds to retrieving the setpoints for the Step and Scan actions. In this example., we assume that retrieving the setpoints from the PrepareResult resource is a first small part of the Step action, but this could also be modelled as a separate action. The ExecuteResult synchronisation resource is used at the end of the activity when information about the execution is retrieved by the Retrieve action.

As the Execute activity also includes information that is not present in the input TMSC model, the Execute activity can partially be generated, but requires manual extension using domain knowledge. This involves the claim and release of the PrepareResult and ExecuteResult synchronisation resources and their dependencies to the synchronisation bars S3, S4 and S5.

```
activity Execute {
  actions {
    ClaC: claim Cyber
    RelC: release Cyber

    ClaP: claim Physical
    RelP: release Physical

    ClaR: claim PrepareResult
    RelR: release PrepareResult

    ClaE: claim ExecuteResult
    RelE: release ExecuteResult

    Retr: Cyber.c.Retrieve
    Step: Physical.p.Step
    Scan: Physical.p.Scan
  }

  action flow {
    ClaP -> |S3
    ClaR -> |S3
    |S3 -> Step
    |S3 -> RelR
    Step -> Scan
    Scan -> |S4
    ClaC -> |S4
    ClaE -> |S4
    |S4 -> RelP
    |S4 -> Retr
    Retr -> |S5
    |S5 -> RelC
    |S5 -> RelE
  }
}
```

Figure 3.10: Textual representation of the Execute activity

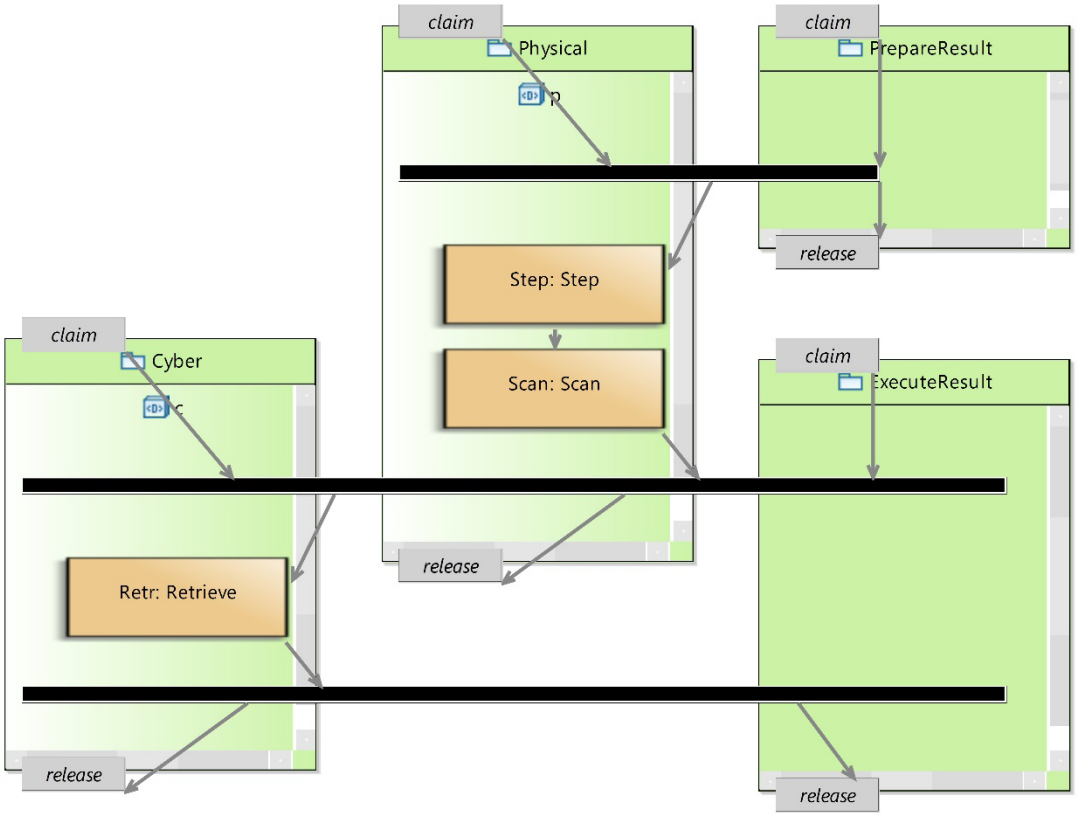

Figure 3.11: Graphical representation of the Execute activity

In Section 3.3.1, we introduced two instances of the parameterised PrepareResult and ExecuteResult. These resources can be seen as buffers between the activities: one activity putting results in the buffer, the other taking them out. The fact that there are two instances of both synchronisation resources is not visible in the Prepare and Execute activities. The activities instances of these resources, but do not specify which one. In fact, both activities are parameterised; they can use both instances of the synchronisation resources. Which synchronisation resource instances are used must be specified in the dispatching specification. This is explained in Section 3.3.4.

3.3.4 Dispatching specification

A dispatching specification specifies the order in which activities are dispatched. LSAT activities synchronise via their shared resources.[3] The claim of a resource by an activity can only happen after the preceding activities have released it. Section 3.3.3 shows that the Prepare and Execute activities share three resources: Cyber, PrepareResult and ExecuteResult. Resource Physical is only used by the Execute activity.

To obtain the same recurring execution pattern as in the TMSC in Figure 3.4, we first dispatch two Prepare activities and then alternately dispatch Execute and Prepare activities. The corresponding dispatching specification for ten exposures is shown in Figure 3.12.

[3] Activities may also synchronise via events, but we have decided not to use events in the transformation from PPS to LSAT.

In this dispatching specification, the parameters between square brackets indicate which instances of the PrepareResult and ExecuteResult resources are used. Between round brackets, the die being prepared and exposed is specified. Note that both the Prepare and the Execute activities alternate between the instances of the synchronisation resources.

```
import "PPS2LSAT.activity"

activities {
  Prepare[PrepareResult.1, ExecuteResult.1] (die=1)
  Prepare[PrepareResult.2, ExecuteResult.2] (die=2)
  Execute[PrepareResult.1, ExecuteResult.1] (die=1)
  Prepare[PrepareResult.1, ExecuteResult.1] (die=3)
  Execute[PrepareResult.2, ExecuteResult.2] (die=2)
  Prepare[PrepareResult.2, ExecuteResult.2] (die=4)
  Execute[PrepareResult.1, ExecuteResult.1] (die=3)
  Prepare[PrepareResult.1, ExecuteResult.1] (die=5)
  Execute[PrepareResult.2, ExecuteResult.2] (die=4)
  Prepare[PrepareResult.2, ExecuteResult.2] (die=6)
  Execute[PrepareResult.1, ExecuteResult.1] (die=5)
  Prepare[PrepareResult.1, ExecuteResult.1] (die=7)
  Execute[PrepareResult.2, ExecuteResult.2] (die=6)
  Prepare[PrepareResult.2, ExecuteResult.2] (die=8)
  Execute[PrepareResult.1, ExecuteResult.1] (die=7)
  Prepare[PrepareResult.1, ExecuteResult.1] (die=9)
  Execute[PrepareResult.2, ExecuteResult.2] (die=8)
  Prepare[PrepareResult.2, ExecuteResult.2] (die=10)
  Execute[PrepareResult.1, ExecuteResult.1] (die=9)
  Execute[PrepareResult.2, ExecuteResult.2] (die=10)
}
```

Figure 3.12: LSAT dispatching specification

3.4 Timing analysis

From the dispatching specification in Figure 3.12, one can perform a timing analysis. This results in a Gantt chart, which is shown in Figure 3.13. The actions in the Gantt chart are coloured according to the parameterised activity that they belong to. Note that the Gantt chart is very similar to the one in Figure 3.4, i.e. the one from the source TMSC. The main differences are the swim lanes corresponding to the instances of the PrepareResult and ExecuteResult resources; these are not in the input TMSC.[4]

[4] The Gantt chart visualisation of LSAT can be adapted to not show the synchronisation resources. Then the Gantt charts would look even more similar.

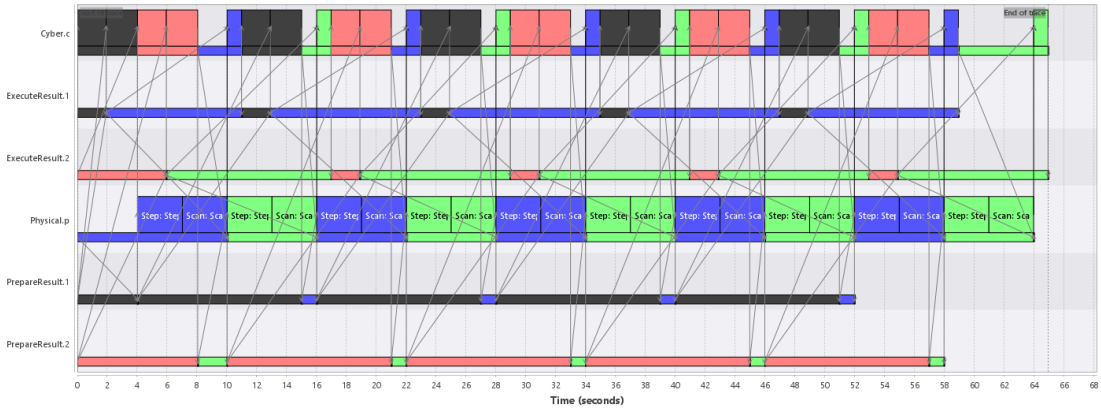

Figure 3.13: Gantt chart from LSAT timing analysis

When applying LSAT's critical path analysis, one obtains the Gantt chart shown in Figure 3.14. The actions that are coloured red are part of the critical path. The Gantt chart shows that the actions of the Execution activity are critical. Only at the beginning of the Gantt chart there are two critical actions from the Prepare activity, i.e. the preparation of the first exposure.

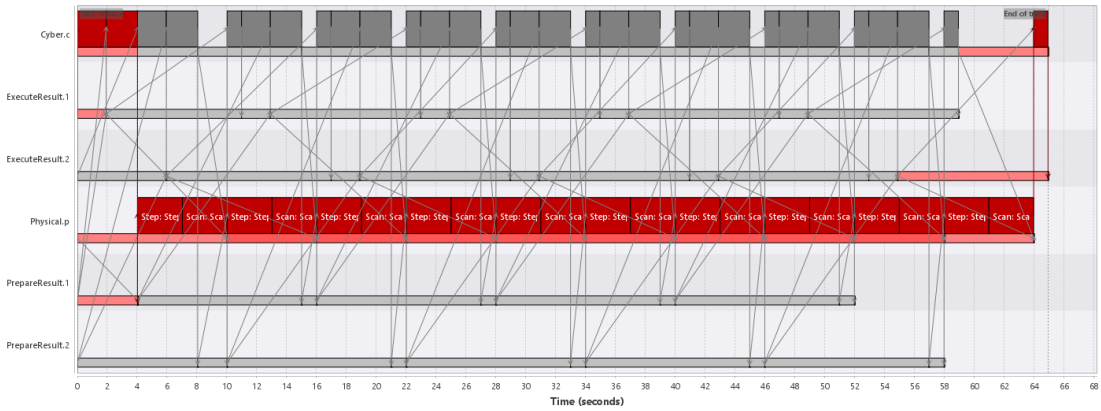

Figure 3.14: Gantt chart from LSAT critical path analysis

3.5 What-if analysis

Having an LSAT model allows exploration of possible future system specifications. One can quickly predict the system behaviour in case of changes. Using the LSAT model created in Section 3.3, one can, for instance, analyse the effect of having a 1-place buffer instead of a 2-place buffer. This corresponds to using only one of the instances of the synchronisation resources PrepareResult and ExecuteResult. The Gantt chart in Figure 3.15 shows that this would hamper productivity: the Execute activity must wait for the Prepare activity to finish.

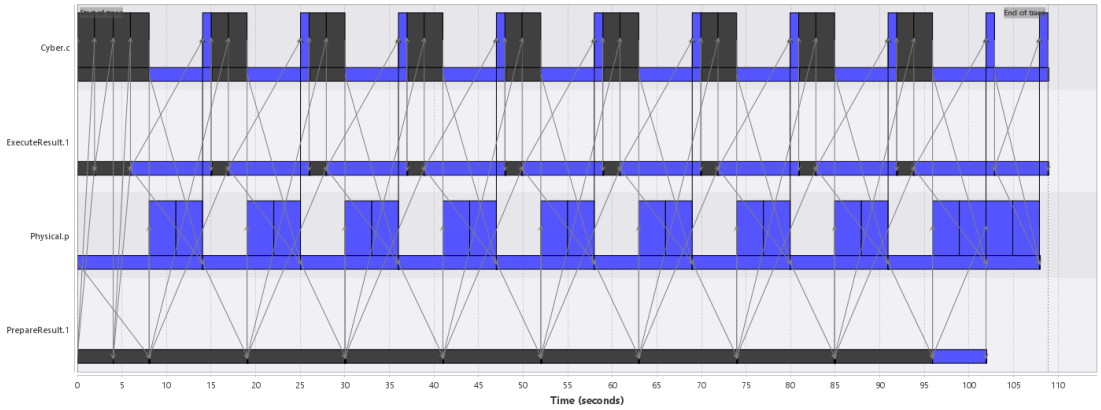

Figure 3.15: Gantt chart from LSAT's timing analysis for what-if scenario with 1-place buffer

This is confirmed by the result of LSAT's critical path analysis for this scenario. Figure 3.16 shows that all actions of the Prepare activity and (nearly) all actions of the Execute activity are critical.

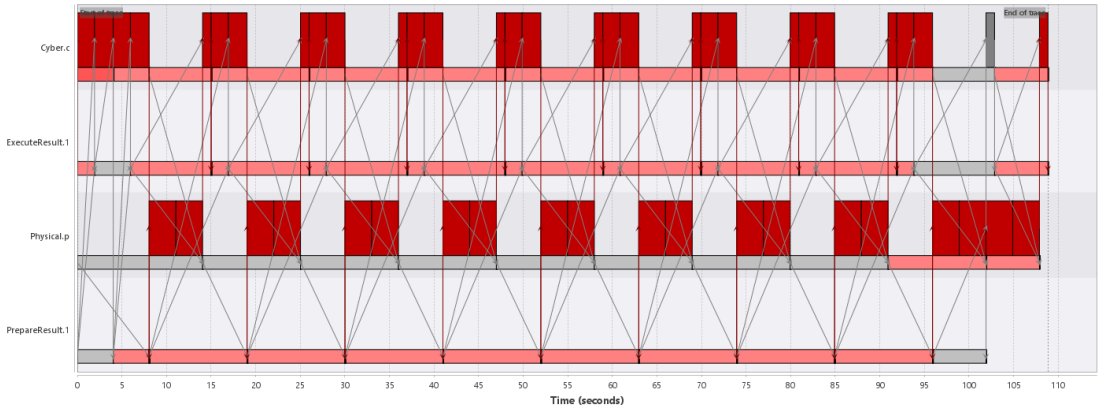

Figure 3.16: Gantt chart from LSAT's critical path analysis for what-if scenario with 1-place buffer

The model also allows what-if analyses with respect to shorter or longer action durations, the introduction of new actions, and the introduction of additional resources.

3.6 Summary

In this chapter, we have illustrated the creation of an LSAT model from an TMSC. The TMSC pre-processing explained in Section 3.2 and the model creation explained in Section 3.3 were both done manually, but as "mechanically" as possible. We draw the following conclusions:

- *Pre-processing*: The TMSC pre-processing explained in Section 3.2 could have been automated by implementing custom rules capturing LSAT model characteristics. These rules would be specific for a family of similar TMSCs.
- *Machine specifications*: The rules explained in Section 3.3.1 are generic. By implementing and applying these rules, most of an LSAT machine specification can be generated automatically:
  - Resources and peripherals can be generated from a TMSC's swim lanes,
  - Peripheral actions can be generated from the different functions performed in an TMSC's swim lanes,

  - Synchronisation resources can be generated from dependencies that are cut by activity generation. Instances of synchronisation resources need to be introduced manually.
- *Setting specifications*: Section 3.3.2 explained rules to create LSAT setting specifications from a TMSC. These rules are generic as well. Implementing and applying them would allow the fully automatic generation of a setting specification, because synchronisation resources do not involve any actions.
- *Activity specifications*: Generating LSAT activity specifications is more challenging. As explained in Section 3.3.3, activities can be partially generated automatically. This requires a specification of the (recurring) actions in a TMSC which together comprise an activity. Using such a specification, the parts that can be automatically generated are:
  - The claims and releases of the used resources,
  - The actions performed in the activity, and
  - The dependencies between these claims, releases and actions including the synchronisation bars needed to obtain a valid LSAT activity.

  The elements of an activity that cannot be generated automatically involve domain knowledge:
  - The claims and the releases of synchronisation resources for synchronisation between activities, and
  - The dependencies between claims and releases of synchronisation resources and actions of regular resources.
- *Dispatching specifications*: Dispatching specifications are the most difficult elements of an LSAT model when it comes to automatic generation. This is because they require the most domain/application knowledge. We conclude that such specifications should be created manually.

We conclude that a large part of an LSAT model can be automatically generated from an TMSC model. The size of this part depends mainly on the number of synchronisation resources needed. If this number is small (compared to the number of swim lanes in a TMSC), then we expect that at least 75% of an LSAT model can be generated automatically from a TMSC.

4 Conclusion

In this report, we have presented a way to start a model-based way of working in case of brownfield development. Instead of manually creating (initial) analysis models, one can derive (partial) analysis models from system artefacts.

We have presented an example to transform a system's execution log into a timing analysis model. Using domain and application knowledge, an execution log captured in TMSC in PPS can be semi-automatically transformed into an LSAT specification and analysis model. From the example, we assess that at least 75% of an LSAT model can be derived automatically from a TMSC in PPS; the rest must be added manually.

Note that the automatic generation percentage depends on the commonality of the input and output formalisms. More similar formalisms are likely to allow a higher degree of automatic transformation than dissimilar ones.

5 References

[1] K. Triantafyllidis, “PPS: From Methodology to Application in Industry,” TNO-ESI, Eindhoven, 2022.

[2] TNO, “Platform Performance Suite (PPS),” 2023. [Online]. Available: https://tno.github.io/PPS/.

[3] B. van der Sanden, Y. Blankenstein, R. Schiffelers and J. Voeten, “LSAT: Specification and Analysis of Product Logistics in Flexible Manufacturing Systems,” in *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)*, Lyon, 2021.

[4] Eclipse Foundation, “Eclipse LSAT,” 2023. [Online]. Available: https://eclipse.dev/lsat/.

[5] R. Jonk, J. Voeten, M. Geilen, R. Theunissen, Y. Blankenstein, T. Basten and R. Schiffelers, “Inferring Timed Message Sequence Charts from Execution Traces of Large-scale Component-based Software Systems,” TU/e, Eindhoven, 2019.

ICT, Strategy & Policy

High Tech Campus 25
5656 AE Eindhoven
www.tno.nl

TNO innovation for life