Safe Behavior Model Synthesis: From STPA to LTL to SCCharts

Jette Petzold[a] and Reinhard von Hanxleden[b]
Department of Computer Science, Kiel University, Kiel, Germany

Keywords: STPA, LTL, Behavior Model, Model Synthesis.

Abstract: In Model-Driven Engineering developers create a model of the system. Typically, such a model is verified to be safe by using model checking. For this, the developers need to create Linear Temporal Logic (LTL) formulas. Determining these formulas and modeling the system in the first place is time consuming and error-prone. We propose to automatically create the LTL formulas based on a risk analysis that has to be done anyway. This reduces errors and the time needed to create the formulas. Furthermore, we use these formulas to automatically synthesize a behavior model of the analyzed system that is safe by construction. The presented approach is implemented in the open-source tool PASTA. A case study with a simplified Adaptive Cruise Control system shows the applicability of the Safe Behavior Model synthesis.

1 INTRODUCTION

Model-Driven Engineering (MDE) is widely used in system development. The developer creates a *behavior model* of the system, which is used as a basis for implementing the system. The safety of the resulting system must be verified e. g. with model checking (Clarke, 1997). When using model checking, the system specifications are translated into formulas using e. g. Linear Temporal Logic (LTL) or Signal Temporal Logic (STL) and a model checker determines whether the model fulfills them. We focus on LTL.

Creating LTL formulas and modeling a system in the first place are non-trivial. System modeling is very time-consuming, which is why techniques exist to generate them automatically (Uchitel et al., 2001). However, with this approach the necessity of creating LTL formulas and verifying the model, still exists. Generating the LTL formulas automatically can reduce the time effort and the risk of mistakes.

To generate the LTL formulas automatically, the safety properties of a system must be determined. This can be done by performing a risk analysis, which is mandatory for safety-critical systems. System-Theoretic Process Analysis (STPA) is a risk analysis technique that identifies more risks than traditional hazard analysis techniques (Leveson, 2016).

LTL formulas can be used to verify a model for the system. However, such a verification is time-consuming. Using the formulas to create a safe by construction model, reduces this time-effort.

This is the reason, we want to use STPA to automatically create LTL formulas and these formulas to automatically synthesize a behavior model that is safe by construction. For that, we propose rules to automatically translate the resulting safety properties of STPA to LTL formulas ensuring that no property is forgotten and reducing the time effort for creating the formulas. Based on these formulas, we propose a synthesis of a Safe Behavior Model (SBM) core as a statechart. This way, the resulting model is safe by construction in respect to the safety properties identified. The presented approach is implemented in the open-source Pragmatic Automated System-Theoretic Process Analysis (PASTA) tool[1] (Petzold et al., 2023) using Sequentially Constructive Statecharts (SCCharts) (von Hanxleden et al., 2014).

Contributions & Outline. section 2 reviews STPA, Statecharts, and LTL formulas. We introduce a running example in section 3. Our main contributions are as follows:

- We propose a translation of the STPA results to LTL formulas (section 4).

[a] https://orcid.org/0000-0002-5559-7073
[b] https://orcid.org/0000-0001-5691-1215

[1] https://github.com/kieler/stpa

Petzold, J. and von Hanxleden, R.
Safe Behavior Model Synthesis: From STPA to LTL to SCCharts.
DOI: 10.5220/0013091600003896

In *Proceedings of the 13th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2025)*, pages 133-140
ISBN: 978-989-758-729-0; ISSN: 2184-4348

- We present an approach to create an SBM core based on these formulas (section 5).
- We extend STPA with Desired Control Actions (DCAs) to cover system goals (section 6).

We present the results of the application of our approach to the running example in section 7 and section 8 discusses the approach. section 9 reviews related work and section 10 concludes the paper.

2 BACKGROUND

We give a short introduction of STPA, followed by the used statechart definition and LTL operators.

System-Theoretic Process Analysis. STPA is a hazard analysis technique consisting of four steps (Leveson and Thomas, 2018).

The control structure modeled in Step 2 consists of controllers and controlled processes. A controller sends *control actions* and consists of a process model that consists of *process model variables* containing information about the system.

In Step 3, the analyst defines *Unsafe Control Actions (UCAs)* by determining in which contexts a control action is hazardous. The context can be stated formally by using context tables (Thomas, 2013). The context is defined by assigning values to the process model variables. For each context the analyst determines whether a control action is hazardous for any UCA type. A UCA can have one of six types. The basic ones are PROVIDED and NOT-PROVIDED. They state that (not) providing the control action is hazardous. Further types are: TOO-LATE, TOO-EARLY, APPLIED-TOO-LONG, and STOPPED-TOO-SOON.

Statecharts. Statecharts are Extended FSMs (EFSMs) that are extended with hierarchy and concurrency (Harel, 1987).

An *execution trace* is a sequence of states, and in- and outputs: $((x_0, s_0, y_0), (x_1, s_1, y_1), \dots)$ with s_i state, x_i input, and y_i output in reaction i.

Linear Temporal Logic. We use LTL formulas for EFSM traces as defined by Lee and Seshia (Lee and Seshia, 2017), especially the following operators: G, $\mathcal{F}$, X, U, and $\mathcal{R}$.

3 ADAPTIVE CRUISE CONTROL EXAMPLE

To demonstrate our approach, we use a simplified Adaptive Cruise Control (ACC) (Venhovens et al., 2000). The vehicle with the ACC follows another one and should accelerate when the distance is above some safe distance, should decelerate when the distance is below the safe distance, and should fully stop when it is below a minimal distance.

We performed STPA on this system. The control structure consists of a software controller with *accelerate*, *decelerate*, and *stop* as control actions. In the process model of the controller, the variable *dist* is defined with the values it can take: *stopDist*, *brakeDist*, *safeDist*, and *accDist*.

We will use the following selected UCAs of the ACC to explain our approach.

UCA1. *Accelerate* is provided when the distance is the breaking distance.

UCA2. *Accelerate* is applied too long when the distance is above the safe distance.

UCA3. *Decelerate* is not provided when the distance is the breaking distance.

UCA4. *Decelerate* is stopped too soon when the distance is the breaking distance.

UCA5 *Decelerate* is provided too late when the distance is the breaking distance.

The context for UCA2 is `dist=accDist` and for the other UCAs it is `dist=brakeDist`

4 FROM STPA TO LTL

To create a model that is safe by construction, first the safety properties must be determined. We use the result of the risk analysis that is done beforehand to automatically generate LTL formulas representing the safety properties. This reduces the time to create such formulas. In STPA the safety properties are the negation of the UCAs. Hence, we use the UCAs for the generation of the LTL formulas. To automatically translate the UCAs, they have to be defined with context tables. Otherwise, the context is stated too informal to automate the process. Furthermore, we need to formalize the meaning of the UCA types. No official interpretation of these types exist. Hence, in this paper we use our interpretations to create corresponding LTL formulas. Abdulkhaleq et al. (Abdulkhaleq and Wagner, 2016) also propose LTL formulas for UCAs, however we disagree with most of their formulas. We discuss this further in section 9.

In the following we use an arbitrary UCA with control action A and context C. We define subformulas for the context variables v, representing that C is present, and control action a, meaning A is sent. Hence, for UCA1, we set $a :=$ *accelerate is sent* and $v :=$ *dist* = *brakeDist*. We will use a short form of the trace definition: We use (v,a) or $(\neg v,\neg a)$, meaning the input and internal variables are set according to v or $\neg v$, respectively. The variables that do not occur in v can have any value, the state s can be any state, and the output must (not) contain A. Figure 1 gives an overview of the traces we reject as unsafe.

Provided Formula. For PROVIDED UCAs we use the formula proposed by Abdulkhaleq et al.:

$$\mathrm{G}(v \rightarrow \neg a) \tag{1}$$

It states that every time the context holds, the control action is not sent. For UCA1 this ensures that *accelerate* is not sent when the distance is the breaking distance. Traces leading to the UCA as seen in Figure 1a violate the formula.

Not-Provided Formula. UCAs of type NOT-PROVIDED can be interpreted in two ways. Either in every reaction where the context holds the control action must be sent or during the timespan where the context holds continuously, the control action must be sent at least once. Since the first interpretation also covers the types TOO-LATE and STOPPED-TOO-SOON, we propose a formula for the second interpretation.

For UCA3 the formula must ensure that *decelerate* is sent at least once when the distance is the breaking distance. A trace as shown in Figure 1b where the context changes to false although the control action was not yet sent should evaluate to false. Let

$$\psi := v \rightarrow (a \mathcal{R} v \wedge \mathcal{F} a),$$
$$\chi := \mathrm{G}((\neg v \wedge \mathrm{X} v) \rightarrow \mathrm{X}(a \mathcal{R} v \wedge \mathcal{F} a))$$

then we translate a UCA of type NOT-PROVIDED to the following formula:

$$\psi \wedge \chi \tag{2}$$

In χ the implicant $\neg v \wedge \mathrm{X} v$ holds in the reactions directly before v changes from false to true. This means the next reaction is the first time where the context holds, and in this reaction $((a \mathcal{R} v) \wedge \mathcal{F} a)$ should hold. $(a \mathcal{R} v)$ ensures that when v changes to false, the control action has to have been sent before.

Since $(a \mathcal{R} v)$ evaluates also to true when the control action is not sent as long as the context holds indefinitely, we ensure with $\mathcal{F} a$ that eventually the control action will be sent. ψ ensures that the UCA does not occur in the first reaction.

Too-Late Formula. For TOO-LATE UCAs we only have to consider the first moment in which the control action should be provided. For UCA5 we want to provide *decelerate* immediately when the distance changes to the break distance. This leads to the following formula:

$$(v \rightarrow a) \wedge \mathrm{G}(\neg v \rightarrow \mathrm{X}(v \rightarrow a)) \tag{3}$$

The second conjunct ensures that traces such as shown in Figure 1c do not occur. The moment the control action should be applied is when the context currently holds and did not hold in the previous reaction. In the formula we capture this in the following way: The control action should be applied in the next reaction if the context currently does not hold and in the next reaction does hold. The first conjunct of (3), namely $v \rightarrow a$, ensures that the UCA does not occur in the first reaction.

Too-Early Formula. We use the following formula for TOO-EARLY UCAs:

$$\mathrm{G}((\neg v \wedge \mathrm{X} v) \rightarrow \neg a) \tag{4}$$

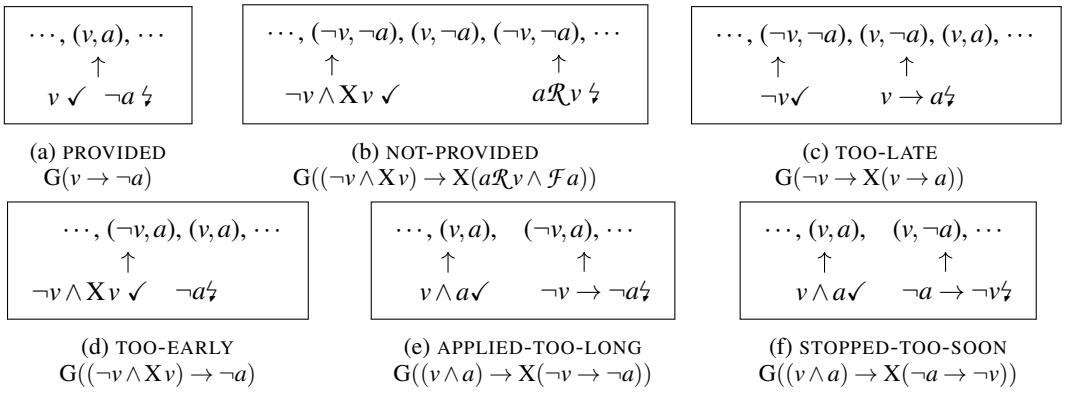
The reaction where the context holds but did not hold in the previous reaction is the first one where the control action is allowed to be sent. Hence, we must ensure that before this reaction the control action is not sent (see Figure 1d). The implicant, namely $\neg v \wedge \mathrm{X} v$, is true if the current reaction is the last one in which v is false before it switches to true. In such a reaction the control action is not allowed to be sent, which is guaranteed by the implication.

Applied-Too-Long Formula. For UCAs of type APPLIED-TOO-LONG we use the following formula:

$$\mathrm{G}((v \wedge a) \rightarrow \mathrm{X}(\neg v \rightarrow \neg a)) \tag{5}$$

To ensure an action is not applied too long, we have to inspect the reactions where the control action is already applied while the context holds $(v \wedge a)$. In these reactions, we must ensure that the control action is not sent anymore when the context does not hold any longer. Hence, we must check whether the context still holds in the next reaction $(\mathrm{X}(\ldots))$. If it does not, the control action must not be sent. Hence, traces such as shown in Figure 1e violate the formula. For UCA2 this ensures that we immediately stop sending *accelerate* when the distance value is no longer equal to the accelerate distance.

Stopped-Too-Soon Formula. To guarantee that a control action is not stopped too soon, we must ensure that after it is sent the first time it is continuously sent



(a) PROVIDED $\mathrm{G}(v \rightarrow \neg a)$

(b) NOT-PROVIDED $\mathrm{G}((\neg v \wedge \mathrm{X}\, v) \rightarrow \mathrm{X}(a \mathcal{R} v \wedge \mathcal{F} a))$

(c) TOO-LATE $\mathrm{G}(\neg v \rightarrow \mathrm{X}(v \rightarrow a))$

(d) TOO-EARLY $\mathrm{G}((\neg v \wedge \mathrm{X}\, v) \rightarrow \neg a)$

(e) APPLIED-TOO-LONG $\mathrm{G}((v \wedge a) \rightarrow \mathrm{X}(\neg v \rightarrow \neg a))$

(f) STOPPED-TOO-SOON $\mathrm{G}((v \wedge a) \rightarrow \mathrm{X}(\neg a \rightarrow \neg v))$

Figure 1: Traces for the different UCA types that are prevented by the presented formulas. Here, we do not consider the first reaction of a trace.

until the context does not longer hold. This means for UCA4 that when we sent *decelerate* in the given context, we must do this until the context changes. We propose a similar formula as for the type APPLIED-TOO-LONG:

$$\mathrm{G}((v \wedge a) \rightarrow \mathrm{X}(\neg a \rightarrow \neg v)) \tag{6}$$

Again, we are interested in the reactions where the context already holds and the control action is applied ($v \wedge a$). In such reactions, the formula ensures that if the control action is not sent in the next reaction, then the context does not hold either. This prevents traces as shown in Figure 1f where the control action is stopped too soon.

5 FROM STPA TO SBM

The generated LTL formulas can be used to perform model checking on the behavior model of the system. These behavior models are usually created manually with the help of supporting tools. However, creating a model tends to be time-consuming and error-prone. When performing STPA before modeling the system, we can use the generated LTL formulas to create the core for a *Safe Behavior Models (SBMs)* that respects the safety requirements by construction. It is not possible to generate a complete SBM since STPA does not contain all relevant information. Business logic not specified in STPA needs to be added manually, which is further discussed in section 8.

We propose an approach for the automatic generation of deterministic SBM cores based on the generated LTL formulas. We assume that no contradicting UCAs exist, which would lead to contradictory LTL formulas. STPA tools such as PASTA help to identify and resolve contradicting UCAs. Additionally, we assume for simplicity that only one control action is sent at a time and that we have a boot-up step to set up the system.

In the following we present a translation for each UCA type. For each translation we use the model resulting from the previous translations as starting point to produce a more compact SBM.

Basic Structure. We start with states that represent a control action each and an initial state representing that no control action is sent. For our ACC example this results in four states: s_{acc}, s_{dec}, s_{stop}, and s_0.

The input variables consists of the process model variables and the actual variables used for their values. In our example we only have the process model variable *dist* with values *minDist* and *safeDist*. Hence, our input variables are *dist*, *minDist*, and *safeDist*. We track the control action that is sent in an output variable named *controlAction*. Each state defines an entry action setting *controlAction* to the represented value.

Not-Provided Transitions. The translation of LTL formulas for NOT-PROVIDED UCAs can be seen in Figure 2. We cannot automatically determine at which moment the control action should be sent. Thus, we set this moment to the first reaction where v holds, which also covers the TOO-LATE type. Hence, we must ensure that if the context holds, we go to the state representing the control action. For each formula we add transitions from the states not representing a to the one representing it with trigger v. Thus, for UCA3 we add transitions from every state to s_{dec}.

Too-Late Transitions. Since we have a boot-up step, we only consider the second conjunct of the TOO-LATE formula (see (3)). We must ensure that we are in s_a when v is $true$. Thus, we add transitions

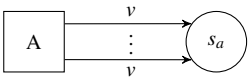


Figure 2: Translation of UCA type NOT-PROVIDED. *A* represents the set of states without the state s_a.

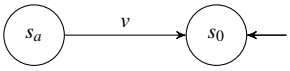


Figure 3: Translation of UCA type PROVIDED.

from states not representing a to s_a with trigger v as done for NOT-PROVIDED UCAs.

Provided Transitions. The translation of PROVIDED UCAs is depicted in Figure 3. We must ensure that if the context holds, the control action is not sent. Hence, we add a transition from s_a to s_0 with the context as trigger. For UCA1 we add a transition from s_{acc} to s_0 with trigger $dist > minDist \wedge dist < safeDist$. For the same context or a context containing v a UCA of type NOT-PROVIDED or TOO-LATE may have been defined. In the first case, we already have a transition that leaves s_a when v holds. In the second case, we have to ensure that the trigger does not evaluate to true if another transition is true.

Too-Early Transitions. TOO-EARLY LTL formulas (see (4)) cannot be translated. We cannot create a transition that triggers depending on values in the next reaction. However, this is not problematic. The presented translation rules ensure that control actions are not sent too-early.

Applied-Too-Long Transitions. If an outgoing transition of s_a is triggered for every context $c \neq v$, the APPLIED-TOO-LONG formula (see (5)) is already fulfilled. Otherwise, we need to split s_a into two states by adding s_{a_v}. For UCA2 we add the state $s_{acc\_dist>safeDist}$.

The result of the translation can be seen in Figure 4. The triggers of the original transitions to s_a are updated to only trigger when v does not hold. Additionally, we copy all outgoing transitions of s_a to s_{a_v} except the ones going to other duplicate states and add a transition between the two states with trigger v. Now, s_{a_v} represents that v holds and the control action a is sent.

To ensure that the control action is not sent anymore when the context changes, we add another transition from s_{a_v} to s_0 with trigger $!v$. To avoid nondeterminism the trigger may have to be adjusted such that it only evaluates to true when the other transition triggers are false.

Stopped-Too-Soon Transitions. LTL formulas for UCAs of type STOPPED-TOO-SOON (see (6)) are translated in a similar way (Figure 5). If in s_a no transition is triggered for v, the formula is already fulfilled. Otherwise, we need to split s_a the same way as done for APPLIED-TOO-LONG.

We modify the outgoing transitions of s_{a_v} such that they trigger when v does not hold. This way, the control action is not stopped too soon. Here, we are interested in the transitions from s_a to duplicates of this state. After all necessary duplicate states are created, these transitions are also copied and modified for the states created during the STOPPED-TOO-SOON translation.

6 DESIRED BEHAVIOR

The generated behavior model fulfills the LTL formulas and hence is safe. However, a system should also fulfill its system goals, which is typically not implied by the safety properties and thus not part of the presented model generation. However, we can use the machinery presented so far to achieve that aim as well.

We extend STPA with *Desired Control Actions (DCAs)*. A DCA determines in which context a control action should (not) be sent to fulfill the system goal. DCAs have two types: PROVIDED and NOT-PROVIDED. During that identification of UCAs, the analysts can declare DCAs as well.

These DCAs can be automatically translated to LTL formulas as done for the UCAs, resulting in a safe model that also fulfills the system goals.

7 COMPLETE ACC

We implemented the SBM core generation in the open source tool PASTA. PASTA provides an STPA DSL and allows the user to select a controller for which the SBM should be generated. The SBM core is generated as an SCChart and contains states and variables as presented in section 5. The UCAs and DCAs are translated to LTL formulas as described in section 5 and are added as *LTL Annotations* to the SCChart, which enables model checking. In each state an *entry* action is defined to set *controlAction* to the correct value.

For the ACC example the resulting SBM is shown in Figure 6. It contains four states: one for each control action and an initial state.

We used LEGO® Mindstorm®[2] robots to test the

[2]https://lego.com

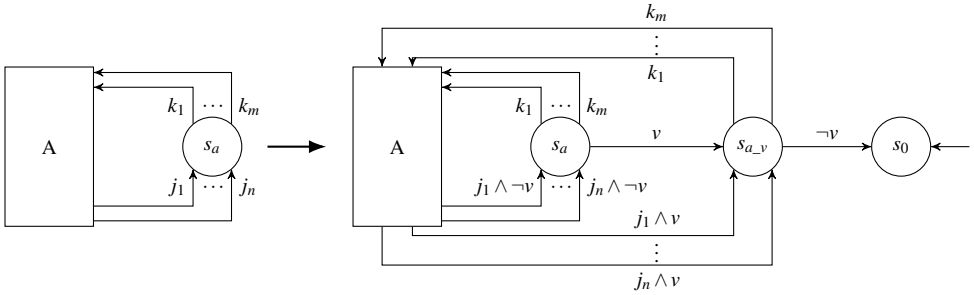


Figure 4: Translation of UCA type APPLIED-TOO-LONG. A represents the set of states without the state s_a.

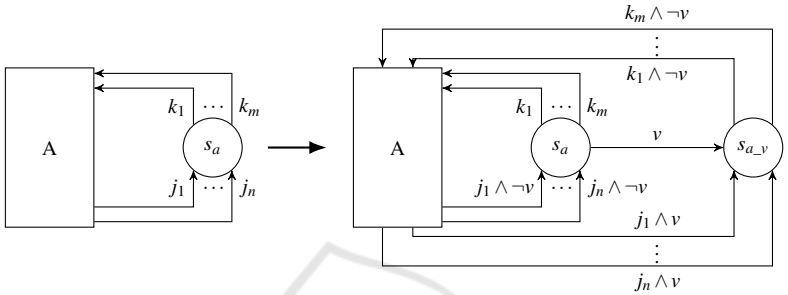


Figure 5: Translation of UCA type STOPPED-TOO-SOON. A represents the set of states without the state s_a.

generated controller[3]. For that, we constructed two robots with wheels and a ultrasonic sensor to resemble two cars. One of them was programmed to drive ahead and change its speed several times. For the other one we performed STPA and used the SBM resulting from our approach as presented before. We added the business logic unspecified in STPA resulting in the SCChart shown in Figure 7. First the values of the sensors are read and `safeDist` is calculated. The *controller* state is the one generated automatically based on the safety analysis. After the controller determines the control action, the new speed is calculated in *newSpeed* and finally the speed is set to the motor. Deploying the software on the Mindstorm® has shown that the controller works as intended.

8 DISCUSSION

We formalized the UCA types since no official interpretation exists. This enables us to automate the generation of the LTL formulas and SBM core. However, other interpretations of the types are possible as well. Whether our interpretation is accepted by the STPA community has to be evaluated in the future.

The resulting SBM core for the ACC example fulfills all generated LTL formulas. Hence, safety properties as well as the system goals are fulfilled. However, the presented synthesis only generates the core of the SBM. Reading sensor values and determining values such as `safeDist` cannot be automatically inferred. The user has to specify this missing information, but the generated SCChart can directly be used for the behavior of the system. When the missing business logic is added, the model depicts the desired and safe behavior since the determination of the control action is not changed by the added information.

We evaluated the approach with further use cases using the LEGO® Mindstorms®. One use case is an ACC that respects a specified desired speed. Another stems from research in the railway domain, where two trains meet on a single track line. They drive towards each other, dock on each other such that passengers can switch the trains, and undock and drive into the opposite direction than before. We performed STPA for these cases and used our approach to generate the SBM. In the second case the SBM consists of eleven states. We added necessary hardware interaction and used the generated SBM for the behavior logic as is. We simulated several scenarios with the Mindstorms® revealing safe behavior.

In conclusion, the generated SCChart core fulfills the LTL formulas used for the generation. The core is a solid foundation for an SBM to which hardware interaction can be added directly, and since risk analysis has to be done either way, the synthesis saves time. Generating the safety properties directly based on a safety analysis can also reduce incorrect or missing formulas.

[3]https://doi.org/10.6084/m9.figshare.27095914.v1

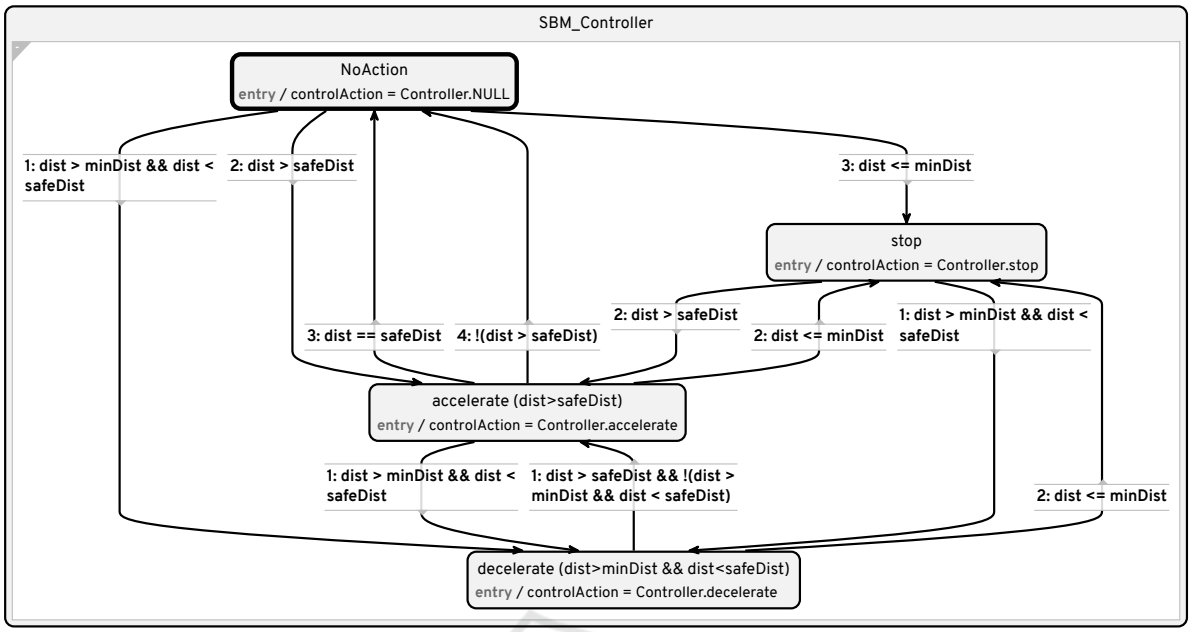


Figure 6: Automatically generated SCChart for the ACC example.

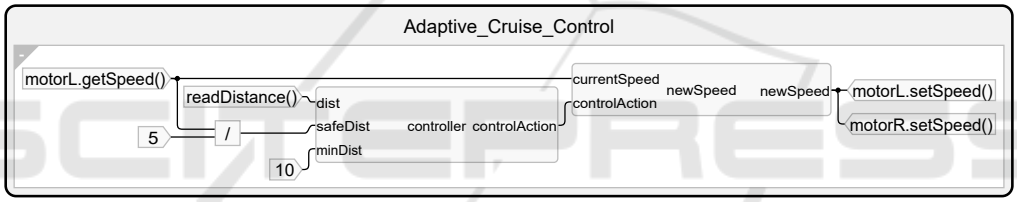


Figure 7: Completed SCChart for the ACC example.

9 RELATED WORK

Abdulkhaleq et al. proposed an approach for creating LTL formulas based on the UCAs of STPA (Abdulkhaleq and Wagner, 2016). We use the same formula they stated for the UCA type PROVIDED. For the NOT-PROVIDED type Abdulkhaleq et al. proposed a formula stating that in every reaction where the context holds the control action must be sent. This interpretation already covers other UCA types, which is why we used our interpretation.

Their formula for type TOO-LATE evaluates to false for some traces in which the control action is not provided too late. Their formula for TOO-EARLY has the same problem. For some traces, where the control action is not sent too early, the formula evaluates to false. The types APPLIED-TOO-LONG and STOPPED-TOO-SOON were not considered in their solution.

Synthesis of behavior models are for example based on Fluent LTL (FLTL) (Giannakopoulou and Magee, 2003) or scenario specifications. Scenarios can be specified with Message Sequence Charts (MSCs), which can be used to synthesize Labelled Transitions Systems (LTS) (Uchitel et al., 2001). We use STPA instead of MSCs. This has the advantage that the results of a risk analysis that must be done anyway can be used. Additionally, this eliminates the problem of combining scenarios in unexpected ways resulting in unexpected system behavior.

When verifying a model according to an LTL formula, the negated formula is translated to a Büchi automaton, the product automaton of the Büchi automaton and the behavior model is built, and the resulting automaton is checked for emptiness (Gerth et al., 1995). In contrast to this procedure, we do not verify or construct a model that fulfills one formula. Instead, we synthesize the behavior model that fulfills all given formulas and can further be used to synthesize the code for software controllers.

10 CONCLUSION

We presented a synthesis of an SBM core from STPA. First we translate the UCAs from STPA to LTL formulas. Then an SBM core is generated by translating each control action to a state and the LTL formulas to transitions. We extended STPA by DCAs to model the desired behavior as well. They can be translated in the same way as UCAs.

The synthesis is implemented in PASTA and creates an SCChart as the SBM. As an example, we performed a safety analysis of a simplified ACC system and deployed the resulting SCChart on LEGO® Mindstorms® showing the intended behavior. Minimal effort is required to specify the interaction with the hardware.

In the future we want to evaluate our approach with complex use cases and whether the DCAs fit within the scope of a safety analyst's work.

ACKNOWLEDGEMENTS

This research has been partly funded by the Federal Ministry for Digital and Transport (BMDV) within the project "CAPTN Förde 5G".

REFERENCES

Abdulkhaleq, A. and Wagner, S. (2016). A Systematic and Semi-Automatic Safety-Based Test Case Generation Approach Based on Systems-Theoretic Process Analysis. *arXiv preprint arXiv:1612.03103*.

Clarke, E. M. (1997). Model checking. In *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17*, pages 54–56. Springer.

Gerth, R., Peled, D., Vardi, M. Y., and Wolper, P. (1995). Simple On-the-fly Automatic Verification of Linear Temporal Logic. In *International Conference on Protocol Specification, Testing and Verification*, pages 3–18. Springer.

Giannakopoulou, D. and Magee, J. (2003). Fluent Model Checking for Event-based Systems. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 257–266.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274.

Lee, E. A. and Seshia, S. A. (2017). *Introduction to Embedded Systems, A Cyber-Physical Systems Approach, Second Edition*. MIT Press.

Leveson, N. and Thomas, J. P. (2018). STPA Handbook. *MIT Partnership for Systems Approaches to Safety and Security (PSASS)*.

Leveson, N. G. (2016). *Engineering a Safer World: Systems Thinking Applied to Safety*. MIT Press.

Petzold, J., Kreiß, J., and von Hanxleden, R. (2023). PASTA: Pragmatic Automated System-Theoretic Process Analysis. In *53rd Annual IEEE/IFIP International Conference on Dependable Systems and Network, DSN 2023, Porto, Portugal, June 27-30, 2023*, pages 559–567. IEEE.

Thomas, J. P. (2013). *Extending and Automating a Systems-Theoretic Hazard Analysis for Requirements Generation and Analysis*. PhD thesis, MIT.

Uchitel, S., Kramer, J., and Magee, J. (2001). Detecting Implied Scenarios in Message Sequence Chart Specifications. *ACM SIGSOFT Software Engineering Notes*, 26(5):74–82.

Venhovens, P., Naab, K., and Adiprasito, B. (2000). Stop and Go Cruise Control. Technical report, SAE.

von Hanxleden, R., Duderstadt, B., Motika, C., Smyth, S., Mendler, M., Aguado, J., Mercer, S., and O'Brien, O. (2014). SCCharts: Sequentially Constructive Statecharts for safety-critical applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, pages 372–383, Edinburgh, UK. ACM.