FORSCHUNGSVEREINIGUNG AUTOMOBILTECHNIK E.V. | FAT-SCHRIFTENREIHE

FAT | 274

Exemplary development & validation of a practical specification language for semantic interfaces of automotive software components

VDA | Verband der Automobilindustrie

Exemplary development & validation of a practical specification language for semantic interfaces of automotive software components

Forschungsstelle:
Lehrstuhl Informatik 11
RWTH Aachen University

Autoren:
Marc Förster
Marko Auerswald
Phillip Keldenich
Stefan Kowalewski

Das Forschungsprojekt wurde mit Mitteln der Forschungsvereinigung Automobiltechnik e. V. (FAT) gefördert.

Abstract

This document reports the results of the FAT/RWTH research project "Semantic interfaces for automotive software components", comprising three work packages.

In work package one description languages used in the automobile industry, for example, EAST-ADL, AUTOSAR and AADL, and related approaches from current research were reviewed. Based on this an initial version of a novel, formal and graphical language for the modelling of component properties was created. The language ("Structured automata"/"STR_UT") is based on modular automata. Structured automata facilitate the expression of propositions about states as well as about events in an interface.

In work package two the initial language was extended with a notion of metric-temporal properties to enable the definition of time intervals and durations. A context-free grammar was defined for structured automata, and the language was integrated into a component model supporting a contract-based, declarative specification. STR_UT specifications are intended to be analysed following the concept of refinement verification, such that the existence of an implementation, as is routinely the case during the early design phase, is not required.

In work package three structured automata were employed to specify two safety and one functional requirement(s) of a Stop/start system model provided by an industry member of the working group. The requirements were refined to the component level and verified by a translation of the STR_UT specification to a network of Timed automata (TA) and its analysis in the TA model checker UPPAAL. For an automatic translation of structured automata to different analysis tools a library was created that presently provides outputs in UPPAAL and Tina (a Time Petri net-based model checking environment) format.

In the case study it was demonstrated

(1) that structured automata can express the necessary formal requirements as well as their refinements,

(2) that EAST-ADL/AUTOSAR timing properties (“TADL constraints”) can be captured without problems,

(3) that STR_UT specifications are more intuitive and more understandable than their TA or TPN (or temporal-logic) counterparts,

(4) that the STR_UT formalisation is flexible, avoiding the rigidity of pattern-based specification techniques, and

(5) that the provided requirements could, after modelling them as structured automata and translating them to a TA network, be analysed by refinement verification in UPPAAL.

Structured automata thus facilitate and improve the creation, verification and reuse of formal specifications as well as that of components and architectures of cyber-physical software in the automotive domain.

Keywords

Cyber-physical systems, model-based development, modular specification, contract-based verification, refinement verification, timed automata, time Petri nets, model checking, behaviour description language.

Table of contents

1 Introduction

The concept of virtual integration of automotive software components already during the design phase requires expressive component specifications that go beyond a merely syntactic characterisation. Application-specific timings have to be captured to predict effects of different assignments of tasks to processors for the behaviour of a complex program. At present, architecture description languages such as AUTOSAR do not, or not sufficiently, consider the problem of semantic component integration. The emerging concept of contract-based, conditional specification enables to discriminate between assumptions of a component on its environment and guarantees the component will deliver upon, given its assumptions. Making this concept practical will enhance the possibilities to reason, in advance, about properties of component assemblies and thus facilitate virtual integration.

This project therefore aims at (1) the development of a language supporting semantic interface descriptions and contractual specification, and (2) its exemplary validation in a case study. The language to develop should be able to express logical as well as metric (for example, timing) properties.

1.1 Research problem

Components are "units of independent deployment" [67]. Accordingly, their specifications should describe them without any knowledge of internals, and the reuse of a (software) component thus becomes synonymous with the reuse of its declarative, black-box specification. For software components to be successfully integrated their composition has to fulfil the requirements of the composite. In traditional development this means that components are tested against their local specifications/requirements before integration, with subsequent integration testing to make sure that higher-level properties still hold.

Unfortunately that is not always the case because those requirements that were tested may have been incomplete.

For similar reasons reuse and replaceability of components in a system is difficult to ensure when it is based upon the presence of a particular implementation. The implementation, in fact, is an implicit specification (we like to style it “operational specification” for that reason). It is unlikely, except for very simple systems, that, formulated in contract terms, the fulfilment of a set of preconditions/assumptions should unconditionally entail fulfilment of the postconditions/guarantees. Only together with an implementation restricting the relationship between inputs and outputs, guarantees will be met:

$$Assumptions \cap Implementation \subseteq Guarantees.$$

This has repercussions on integration: many implementation variants for a given component will epitomise implicit requirements (and, of course, its explicit requirements) such that integration results in a correctly functional system. Another variant may, however, fail to meet those implicit requirements, resulting in a system that does not work as intended even though the component meets all its explicit requirements. Such integration faults can be hard to detect or prevent.

A specification has to be complete or should be, at least, as complete as possible. The difficulty of distributed development, thus, are not just the “incomplete characterisations of the environment of the system to be developed by the supplier” [6] but also unspecified properties that are tacitly, even unknowingly, assumed to be present. If the finished component possesses them all the same there will not be a problem. But an implementation that conforms with some requirements, and together with the rest of the system works fine, is no sufficient evidence for reuse or the possibility of replacement of that component with another one conforming to the same specifications.

This observation gives rise to the concept of inherent verification or “refinement verification” [64], referring to a style of verification that solely relies on declarative specifications, without considering particular implementations, to show the correctness of the intended system (see Section 1.3.4). In this fashion the blind spot of implementation-based verification can be avoided and completeness of specifications assured. This does not obviate testing, and Donald Knuth's dictum still holds, of course: "Beware of bugs in the above code; I have only proved it correct, not tried it." [43] Nevertheless, a proven refinement is a necessary condition for the systematic reuse of cyber-physical software components (CPSWC).

Application of refinement verification hinges on the practicability of formal methods of behaviour specification and verification. The work reported here focusses on two aspects:

- how to specify the interface of a CPSWC in a semantical way. Because the semantics of a piece of software primarily lies in its behaviour, this means specifying that behaviour in a declarative, formal and analysable way.
- to define a corresponding behaviour specification language that is "practical", in the sense of being easily accessible by practitioners while still being fully formal. The language should be usable as a frontend for several, though related, underlying formalisms, not just one.

1.2 Use cases

The following use cases for the specification language and analysis method were identified at the start of the project:

(Z) Exemplary specification of contracts for a component and their refinement to subcomponents of a Stop/start system model provided by an OEM, based on existing verification conditions.

(A) Proof of verification conditions against their refinement to show that the latter could be given to a supplier for the implementation of a subcomponent.

(B) Given two contract refinement levels: what are the effects of changes of implementations/of contracts on one level or the other to the remaining components and the whole? (Merged with use case C.)

(D) Given a system that is specified by a contract and is structured into components: how to find component contracts such that the system contract is fulfilled?

(E) Given component implementations, from which contractual specifications can be derived: how to find "the minimal" contracts of these components such that together they fulfil a specific contract of the system to be built from the components?

1.3 Related work

In the context of inherent/refinement verification we consider related work that is aimed at providing analysable behavioural descriptions of cyber-physical software: architecture description and property specification languages, patterns and tools.

1.3.1 Architecture description languages

Current software ADL provide means to specify a large spectrum of stakeholder concerns, but from their beginnings in the 1990s all of them have had two common objectives: capturing structure in terms of components and their logical relations, and characterising the behaviour of components/composites [46].

AUTOSAR. The AUTOSAR software component template [3] defines a language for the description of software components (SWC). Application components that have been developed against the virtual functional bus (VFB) to execute in the runtime environment (RTE) can be specified with it completely. The description takes place on three distinct abstraction levels (Figure 1):

The **VFB level** specifies communication properties of SWC and their relationship to each other by port interfaces, port prototypes and the data types they use. Port prototypes, for example, can realise a client-server or a sender-receiver behaviour, and they can provide or require a port interface. Port interfaces in turn represent the type of the port prototypes they are associated with, making ports with the same interface compatible by definition. Port interfaces support a design-by-contract paradigm to aid development.

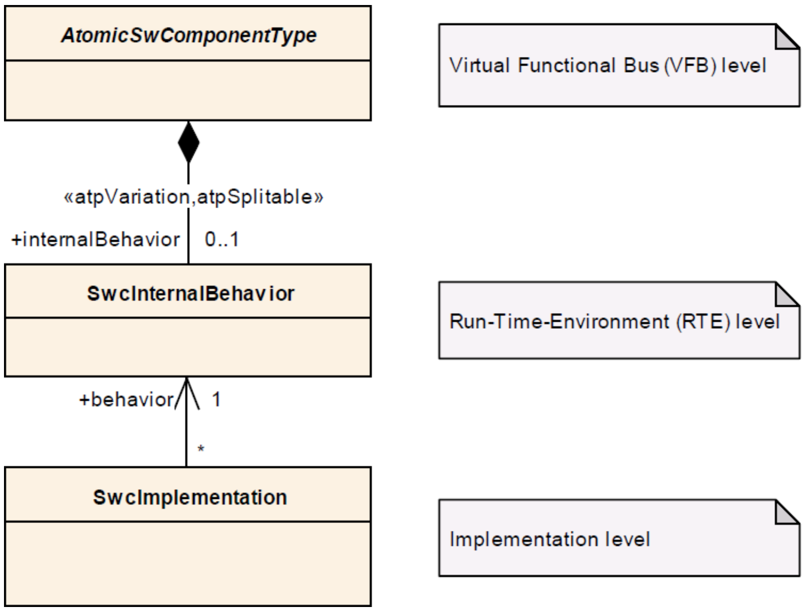


Figure 1. AUTOSAR software component description levels (from [3]).

The **RTE level** provides further means of describing SW behaviour in the runtime context. As an operating system (OS) standard, on the RTE level AUTOSAR views software components primarily as schedulable entities (runnables), featuring RTE events such as data received, data receive error, timing (triggering a runnable), mode switch etc. Runnables are part of atomic software components which can be hierarchically aggregated to software components. An atomic SWC may have multiple internal behaviours which are characterised further by memory areas, shared variables for communication, and their requirements on RTE/OS services. The actual, dynamic behaviour of a SWC is encapsulated in an implementation on this level.

On the **implementation level**, runnables are associated with code. The internal behaviour of an atomic SWC can have multiple implementations, in various programming languages and with different optimisations.

EAST-ADL. Despite having three behaviour abstraction levels, AUTOSAR SWC descriptions are still rather implementation-oriented, lacking, by design, the means to capture, for example, concepts important early on in the design process, such as requirements, vehicle features and the like. EAST-ADL [2] [42] includes AUTOSAR as its implementation level and builds three modelling levels on top of it:

The **vehicle level** is the most abstract. On it, features are specified as they are perceived by the customer. The feature model is derived from top-level requirements and includes feature-specific requirements as well as possible configurations.

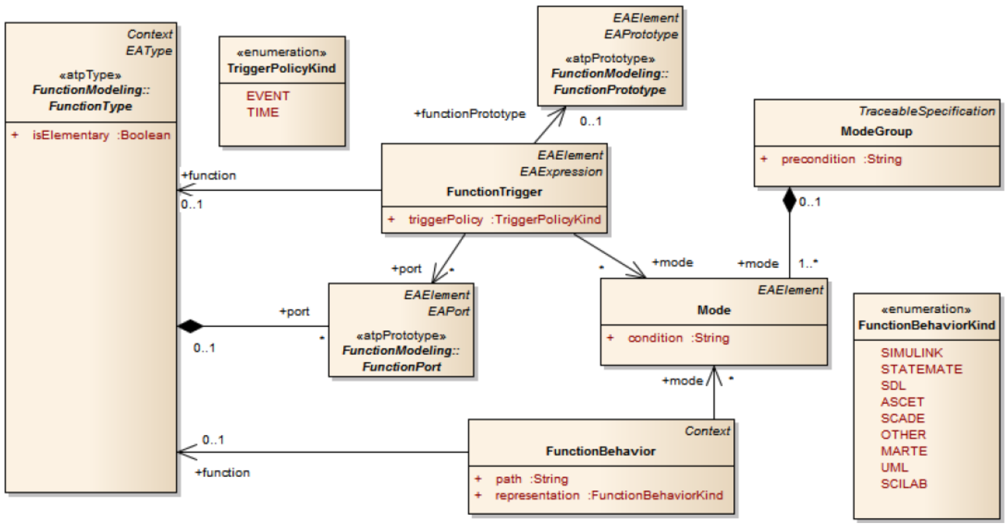


Figure 2. EAST-ADL function behaviour model (from [2]).

On the **analysis level** features and subfeatures are mapped to a functional architecture realising them. Functions, on this level, are collaborating, abstract entities with interface and behaviour specifications, and there is no discrimination between software and hardware. A stop/start feature, for example, is realised by functions comprising gear lever and clutch pedal sensing, engine control etc.

The **design level** decomposes the functional architecture further, already assigning specific functions to software (control, signal transformation) or hardware (sensors, effectors) elements. A corresponding hardware architecture is introduced. SW functions are being detailed in such a way as to be assignable to AUTOSAR software components and runnables. For this purpose EAST-ADL defines associations between ADLFunction entities and AUTOSAR runnables [20]. Functions can be time triggered (in regular intervals) or event triggered (by incoming data or client request, for example). Trigger policies are temporal constraints such as trigger period or they set a condition under which triggering will be successful. The actual realisation of software behaviour is referred to domain-specific tools, Simulink, SCADE etc (Figure 2).

AADL [31]. Originally, the SAE's *Architecture analysis and description language* was named *Avionics architecture description language*. Nevertheless, it has been consistently used in the automotive domain. While EAST-ADL, together with AUTOSAR, is a stack of conceptual, "horizontal", abstractions from vehicle features down to schedulable software entities, AADL is more device and implementation oriented: specifications are organised in terms of processor, memory, bus, data, thread, thread group, process, subprogram etc [47]. Like runnables in AUTOSAR, threads are AADL's schedulable entities.

In contrast to EAST-ADL/AUTOSAR, AADL has behaviour and programming language annexes that define a high-level specification/programming language. The behaviour annex contains several sublanguages: Automata language, Thread dispatch language and Component interaction language. Figure 3 shows the excerpt of the specification of a token ring with thread dispatch, states, events and timing [8]. This thread is implemented as a timed automaton featuring three event ports that trigger internal transitions or are triggered by them; it is scheduled with a fixed period.

Comparing AADL with EAST-ADL/AUTOSAR regarding software specification a salient difference is their treatment of hierarchy. While EAST-ADL takes a top-down view tailored to automotive development, successively decomposing high-level entities, AADL composes low-level entities to programs, devices and, eventually, systems in a bottom-up fashion.

```
thread Node
   features prev: in event port; succ: out event port;
            start: in event port;
   properties Dispatch_Protocol => Sporadic; Period => 10ms;
end Node;

thread implementation Node.i
   annex behavior_specification {**
      states idle: initial complete state;
             wait: complete state;
             cs: state;
      transitions
         idle -[ start? ]-> idle { succ!; };
         idle -[ prev? ]->  idle { computation(3ms); succ! };
         idle -[ prev? ]->  wait { computation(3ms); succ! };
         wait -[ prev? ]->  cs;
         cs   -[ ]->        idle { computation(5ms, 10ms); succ!; };
    **};
end Node.i;
```

Figure 3. AADL thread specification of a token ring node.

In their cores both languages offer a definition of the syntax for relatively abstract property descriptions but rely on additional elements for the specification of properties such as functional behaviour, timing (only AADL has timing included in its core, AUTOSAR has started to do so in version 4) or reliability. These are defined in separate “extensions” or “annexes” (Figure 3 and Section 1.3.2).

1.3.2 Property specification languages

The capabilities of both AADL and EAST-ADL/AUTOSAR are enhanced by several extensions/annexes. While, for example, the AADL core already has a notion of timing, EAST-ADL as well as AUTOSAR provide language constructs for it in their timing extensions. These integrate the concepts developed by the TIMMO project for the Timing-augmented description language (TADL/TADL2) for timing specifications: **TADL** describes timing requirements based on events and event chains [60]. The latter are concatenated to yield so-called end-to-end timing specifications for critical execution paths and the complete data flow between associated sensors and effectors.

Other language extensions/annexes for specialised properties are contained in the error model and behaviour annexes of AADL and in the variability and dependability extensions of EAST-ADL. Nevertheless, even though they characterise the language elements for property description, the corresponding formalisms and analysis methods are still one step further removed and have to be provided by software tools.

AADL2FIACRE, part of the TOPCASED project [8] [9], for example, is a verification toolchain for AADL behaviour specifications: they are translated to Fiacre as intermediate language, and from there to tool-supported formats for analysis (Tina for real-time Petri nets, CADP for process algebra) [8]. Models are checked against requirements formulated in SE-LTL [17]. For a formal analysis of TADL2 timing specifications in EAST-ADL, Goknil & al [26] propose a tool-supported simulation and verification method. The semantics of timing constraints are given by a mapping to the Clock Constraint Specification Language (CCSL). Simulation is carried out directly on the TADL model ("rapid prototyping of TADL2 specifications"). For verification the model is translated to timed automata and checked with UPPAAL [40].

The Structured Assertion Language for Temporal Logic **(SALT)** adapts the hardware verification language **PSL** (Property Specification Language), standardised by the IEEE [45], to use with real-time software [5]. SALT provides a user-friendly frontend to LTL and TLTL specifications, combined with language elements for exceptions, macros and the star-free fragment of regular expressions, which is translatable to LTL. Translators to SMV [34] and SPIN [39] syntax exist, facilitating model checking of SALT specifications.

1.3.3 Specification patterns

Formalisation of requirements for CPSW necessitates intimate knowledge of the formalism used, be it temporal logic, Petri nets, process calculus or similar. A common approach to ease the application of such languages is to have a predefined set of complex expressions in the language that are supposed to cover most properties an engineer wishes to state and verify. Such expressions are called “patterns”.

Consider the following requirement on an elevator:

> *Between the time the elevator is called at a floor and the time it opens its door at that floor the elevator shall arrive at that floor at most twice.* [22]

This seems reasonable to wish for and is easy to understand: the elevator should pass by my floor just once before I can enter it. A translation of the requirement to LTL, however, looks a bit more involved:

$$\boldsymbol{G}\left(\begin{array}{c}(call \land \ \boldsymbol{F}\, open) \rightarrow \\ ((\neg atFloor \land \neg open)\ \boldsymbol{U} \\ (open \lor ((atFloor \land \neg open)\ \boldsymbol{U} \\ (open \lor ((\neg atFloor \land \neg open)\ \boldsymbol{U} \\ (open \lor ((atFloor \land \neg open)\ \boldsymbol{U} \\ (open \lor \big((\neg atFloor \land open))))))))\big)\end{array}\right).$$

The semantics of the LTL formula is less obvious than that of the original, natural-language requirement, and it may take an expert to be sure that it indeed matches the desired property. To address this issue, [22] introduced the Specification pattern system (SPS), defining expressions in structured, parameterised language that model typical requirements on software systems. SPS expressions are meant to be automatically transformed to LTL for model checking.

SPS expresses properties as the combination of a pattern and a scope. The pattern specifies the desired behaviour, and the associated scope states the (logical) time period during which the pattern shall hold. Available patterns can be used to state, for example, the absence or existence of a property, or a particular order of events.

SPS expressions are much easier to deal with than LTL but they are rather inflexible, allowing to combine just one scope with one pattern, parameterised by a single atomic proposition (AP). For this reason SPS was extended with Composite propositions (CP) by [55] to allow more complex specifications. CP were later formalised by [63], enabling the use of combinations of AP such that also entire sequences of states can serve as scope boundaries. [1] extend the patterns approach of [22] to specify real-time requirements. The system is modelled in the Fiacre intermediate language and is checked using Tina [36].

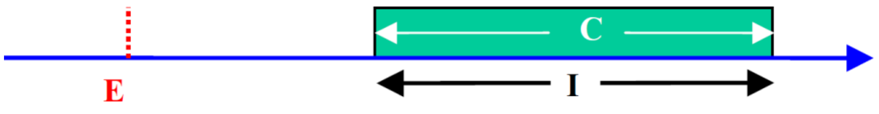


Figure 4. CSL pattern P1: "Whenever [E] occurs [C] holds during following [I]".

The patterns approach to formal specification has also been taken by [15], as well as in the SPEEDS project [38], as a user-friendly frontend to hybrid automata (Extended state machines, ESM) for the Contract Specification Language (CSL) (Figure 4) [24]. An example for the employment of patterns in an industrial verification tool is BTC Embedded Validator [33] where pattern instantiations are translated to safety automata for a CTL model checker (Section 1.3.5).

1.3.4 Inherent verification of requirements

Most model checking approaches presuppose the existence of a system model to check against formal requirements. But whenever the primary aim is to make sure, as it is during the design phase, that the specification and its refinement do make sense, this can be decided by virtue of specifications alone: if the composition of subcomponent specifications implies fulfilment of the superordinated specification, one needs only to verify a particular subcomponent implementation against its local specification to make sure it will work together with the rest of the system as intended [30]. This method of specification integration could be styled "implementationless verification" or "inherent verification". [64] uses this approach (under the name "refinement verification") with higher-order logic, [4] augment inherent verification with techniques to determine coverage as a measure for requirements completeness, based on LTL.[1]

1.3.5 Tools for CPSW behaviour verification

In model-based development, a significant part of the specification language for static properties, behaviour, implementation and formalised requirements depends on the tools employed by the engineers. From the large number of MBD tools in existence we characterise some representatives that integrate formal verification.

	Timing model	***Inherent verification***	***Structuring of declarative specification***	***Multiple instantiation***	***States (S) and/or events (E)***	***Formal basis of specifications***	***Analysis method***
SCADE Design Verifier	Discrete	No	Autom/DFD	Yes	S	LTL	MCh/ SMT
Simulink Design Verifier	Discrete	No	Autom/DFD	No	S	LTL	MCh/ SMT
Simulink + BTC Emb Validator	Discrete	No	Patterns	No	S	CTL/ Autom	MCh
AutoFocus + SALT	Discrete	No	SALT	Yes	S	(T)LTL	MCh
AutoFocus + Isabelle/HOL	Discrete	Yes	FOCUS	Yes	S	HOL	ThProv

Table 1. Characteristics of representative verification tools.[2]

SCADE [35] is based on the synchronous languages Esterel and Lustre. As IDE for embedded systems development it features data flow diagrams for control design and automata (akin to statecharts) for state-dependent

[1] We propose the term "inherent verification" for implementation-free consistency and sanity checking of formal specifications because the names "refinement verification" or "refinement-based verification" have also been used to denote methods that do rely on the presence of implementations or implementation models ([58] [55]).

[2] *Autom*: Automata. *DFD*: Data flow diagrams. *HOL*: Higher-order logic. *SMT*: Satisfiability modulo theory. *MCh*: Model checking. *ThProv*: Theorem proving. *(T)LTL*: (Timed) Linear temporal logic. *CTL*: Computation tree logic.

properties. **SCADE's Design Verifier** can be used to check an operational model against a specification which affords similar language elements as the model. Multiple instantiation of models/modules/components is possible, and the communication between them is value (state) based. Specifications are transformed to LTL, and the model is verified by checking through SAT/SMT solving.

The **Simulink Design Verifier** is similar to SCADE's.[3] A difference from the usability perspective is Simulink's missing namespace concept, thus, models can be instantiated just once. An external plugin called **BTC Embedded Validator** (EV) also offers formal verification of Simulink models. In contrast to both design verifiers mentioned above it features the branching-time logic CTL. BTC EV aims at facilitating its use by a pattern concept. Parameterised requirement templates in structured language are transformed to automata, and the model is checked against them. The Stop/start system that features in this report's case study (Section 3) was verified by its developers using BTC EV.

AutoFocus is a modelling and verification environment for distributed, embedded real-time systems developed by TU München. It is based on the theory of stream-processing functions realised in the FOCUS language [12] and AutoFocus tool [32] (see [62] for a survey of the topic). Behaviours can be specified as (stateful) automata or (stateless) functions. AutoFocus models are used for C code generation and can be verified in two distinct ways. One approach uses **SALT** (Section 1.3.2) to specify properties for checking the model, the other generates **Isabelle/HOL** [44] theories from AutoFocus models of different abstraction levels and performs inherent/refinement verification [64] (see Section 1.3.4) to prove that low-level component specifications indeed fulfil the (formal) system requirements.

[3] For analysis, both SCADE‘s and Simulink‘s design verifier use the *Prover* plugin [37]. BTC Embedded Validator is a commercial frontend to the VIS model checker [41].

1.4 Summary

A common characteristic of available behaviour modelling and analysis tools, patterns and languages, with the singular exception of the combination of AutoFocus and Isabelle/HOL, is that they require an operational specification (implementation) to be present which should be verified against formal requirements. Tools additionally specialise in code generation.

For the development of automotive CPSW there is a trade-off between EAST-ADL/AUTOSAR and AADL: the former are failing to capture application-specific timing aspects [23] due to their OS-centred view on applications as schedulable entities, the latter does provide a complete timing concept but does not offer the conceptual abstractions necessary in a top-down development process from vehicle features down to thread/runnable level. More sophisticated aspects of behaviour modelling are delegated to tools. Constraints can be specified, but contracts are not considered.

To facilitate formal specification of requirements, patterns are a solution that, however, may be hampered by their lack of flexibility. Pattern libraries have been reported to grow beyond manageability ([25] states this for the contract specification language, CSL, of the Rich component model [24]) because of the multitude of variants that users demanded. Dwyer et al conducted a poll [22] collecting in excess of 500 LTL properties from academic and industrial users, claiming that 92% of them could be expressed by one out of a set of eight patterns. The catch is in the fine print because the authors went on to add, rather vaguely: “or a variation of them”.

An approach that is flexible and user friendly at the same time appears to be SALT’s adaptation of PSL as an LTL/TLTL façade. So far, that concept does not include refinement verification, which, as far as tools are concerned, is exclusive for AutoFocus with Isabelle/HOL and theorem proving. Viable techniques relying on a purely declarative specification for verification purposes comprise

- specification with higher-order logic, inherent verification by theorem proving (previously realised by AutoFocus and Isabelle/HOL),
- specification with temporal logic (preferrably with a language such as PSL or SALT), inherent verification by satisfiability checking (not done yet),
- specification with automata, inherent verification by model checking (not done yet).

Theorem proving is a powerful reasoning method and enables the treatment of large, even infinite, state spaces. On the downside it requires a considerable amount of expert and user intervention, potentially making its application difficult in an industrial setting. We would like to enable mechanised inherent verification based on a user-friendly language. Confronted with the choice between TL and automata specifications we opt for automata and give a motivation in the following.

2 An automata-based approach to CPSW specification

CPSW controlling vehicular functions are, typically, hard real time applications. A service being delivered late is synonymous with a service not delivered at all. Thus, in this context timing is not negotiable and ceases to be a “nonfunctional property”. The contract classification of [10] can be adapted accordingly (Figure 5). We also do not hold pre- and postconditions and invariants to be on a separate level since in the automata-based perspective properties on higher levels are specified by them. The basic language elements to be proposed express logic combinations of properties (semantic level), event ordering (pragmatic level) and timing properties (quantitative level).

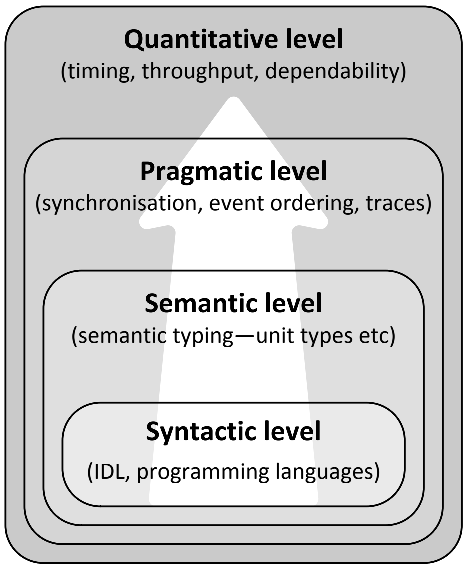

Quantitative level
(timing, throughput, dependability)
Pragmatic level
(synchronisation, event ordering, traces)
Semantic level
(semantic typing—unit types etc)
Syntactic level
(IDL, programming languages)

Figure 5. Levels of contractual specification of cyber-physical software.

Compared with temporal logic, using deterministic, finite automata for property description imposes certain limitations: for example, only safety properties can be checked in this manner because their violation has a finite witness, while liveness properties do not. This is in fact a less serious issue than it might seem at first [13]. Most liveness properties of interest can be transformed to similar safety properties: proving “The brake effector will react within 20 ms.” (safety) is even arguably more interesting than proving “The brake effector will react eventually.” (liveness).

Skipping liveness, restricting consideration to finite traces offers significant advantages: reasoning becomes simpler and model checking is reduced to invariance/assertion checking [53]. Furthermore, such an approach enables bounded and on-the-fly model checking algorithms [19] and incremental language inclusion checks [51], obviating the need to generate the entire state space at once. Another pro that comes with the use of deterministic automata for property specification is their ability to be easily complemented, even when they are timed, which is useful for deriving contract normal forms.

Since automotive CPSW is distributed, from smaller SoC to a full-blown AUTOSAR-compliant product, a specification language must not presuppose a global timebase. Furthermore, multi-form time [7] will be useful when component dynamics critically depends on processor cycles, crankshaft turns etc. To align it with ADL concepts (“RTEEvent”), states and events should be equally treated.

Last but not least, there is the aspect of usability: the language should be able to cover a variety (though related) of formalisms, variants thereof and analysis techniques. A graphical, block-diagram-like notation that is familiar to engineers is desirable, as well as an optional programming-language “skin”. This report proposes such a notation, based on standardised fragments in the form of modular TPN (or, equivalently, networks of TA).[4]

2.1 Traces & contracts

Our approach to CPSW specification is based on timed traces, assertions on traces and *invariants* realised by automata that are timed, relating

[4] There exists a multitude of modularity concepts for Petri nets (see [54] for a survey, and [50] for a formalisation aimed at a standard for modular PN), as well as the formalism of networks of timed automata as it is realised in UPPAAL, for example. All of these, however, define the PN/TA fragments on the level of primitives (edges and vertices) alone.

assertions. This is akin to the concept of stream-processing functions, as introduced by [14] and further developed by [10] (Section 1.3.5).

Invariants being a central concept of STR_UT, it is in order to give a brief definition:

$$
\begin{aligned}
&{}^{X}Interface = \bigcup_{i \in I} {}^{X}Port_i \\
&Domain\left({}^{X}Port_i\right) \overset{\text{def}}{=} {}^{X}DP_i \\
&{}^{X}Proposition_i : {}^{X}DP_i \mapsto \mathbb{B} \\
&Time : T \mapsto {}^{X}DP_i \qquad\qquad (T \subseteq \mathbb{N}, \mathbb{Q}_0^+, \mathbb{R}_0^+) \\
&{}^{X}Evaluation_i : \left[{}^{X}DP_i\right]^{|T|} \mapsto \mathbb{B}^{|T|} \\
&{}^{X}Invariant : \bigcup_{i \in I}\left[{}^{X}DP_i\right]^{|T|} \mapsto \mathbb{B}^{|T|}
\end{aligned}
$$

The interface of a component X is the set of its ports. The domain of a port is its associated trace, which is a succession of timed values over a domain. Time can be discrete or continuous. A proposition, pertaining to a port, assigns *true* or *false* to every value of the trace of its port. An evaluation function maps the trace to a Boolean trace, according to its proposition. An invariant, in turn, does the same for all ports of the interface.

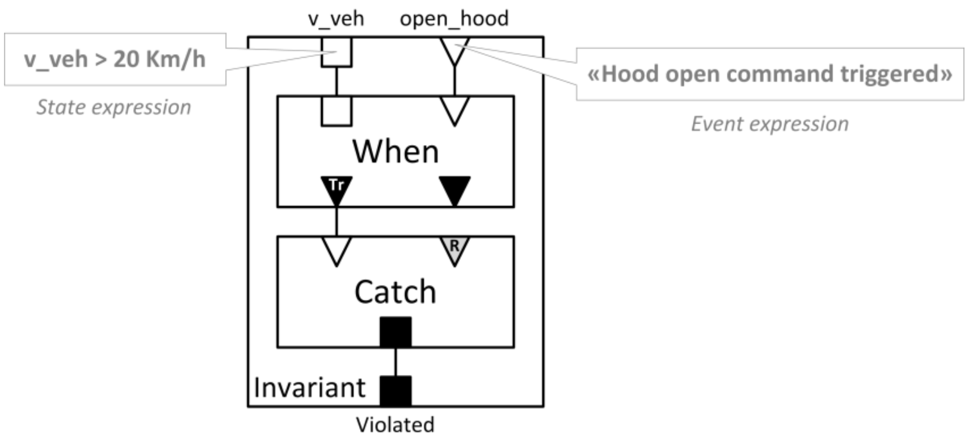


Figure 6. A safety requirement of a convertible car.

Figure 6 shows the example of an invariant in STR_UT notation, relating two propositions: a state expression (“v_veh > 20 Km/h”) and an event expression (“Hood-open command triggered”). It asserts that no open-hood command shall be triggered while the (convertible) vehicle is moving faster than 20 Km/h. Note that, although invariants are theoretically functions of all ports (respectively, of their propositions), some ports may be ignored by it just like a don’t-care argument of a Boolean function. In the STR_UT setting, assumptions and guarantees of contracts are expressed by means of invariants.

To perform a refinement proof on a contract $C = (A; G)$ and its decomposition in subcomponent contracts $C_1 = (A_1; G_1)$, $C_2 = (A_2; G_2)$, …, $C_n = (A_n; G_n)$ one has to show

- that the superordinated contract is satisfied whenever its subcontracts hold, and
- that the assumption of the contract of each subcomponent is satisfied whenever the contracts of all other subcomponents hold, given the superordinated assumption.

In mathematical terms, it must be proved that [18] [21]

$[\bigwedge_{1\leq i\leq n}(A_i \rightarrow G_i)] \rightarrow (A \rightarrow G)$ and

$[A \wedge \bigwedge_{2\leq i\leq n}(A_i \rightarrow G_i)] \rightarrow A_1$ and

$\left[A \wedge \bigwedge_{1\leq i\leq n, i\neq 2}(A_i \rightarrow G_i)\right] \rightarrow A_2$

and …

$\left[A \wedge \bigwedge_{1\leq i\leq n, i\neq j}(A_i \rightarrow G_i)\right] \rightarrow A_j$

and …

$[A \wedge \bigwedge_{1\leq i\leq n-1}(A_i \rightarrow G_i)] \rightarrow A_n$.

Additionally, implications must not be vacously true, so the composition of subcontracts, and thus the system conforming to those contracts, has to be realisable. Note that the supercomponent can be specified by more than one contract each of which will typically require different sets of subcontracts to prove it. If both the superordinated contract and its subcontracts are unconditional guarantees (as is the case with the safety requirements of the case study, Section 3.2) the proof reduces to

$[G_1 \wedge G_2 \wedge \ldots \wedge G_n] \rightarrow G.$

With this, a tentative answer to use case B/C (Section 1.2) may be sketched thus:

- If a superordinated guarantee G changes and its fulfilment cannot anymore be derived from the subguarantees it depends upon, one or more of these have to be adapted or additional ones introduced. Existing subcomponent implementations affected by new or changed subguarantees must be rechecked against their superordinated contract(s)/guarantee(s) and, if necessary, adapted.
- When a superordinated guarantee is newly introduced it may be possible to prove its satisfaction using existing subguarantees. If this cannot be done subcontracts must be changed/introduced and existing implementations rechecked and potentially adapted.
- When a subguarantee G_i changes, the refinement of all superordinated guarantees depending that subguarantee for their proof has to be rechecked. If a superordinated guarantee should fail the check either it or one or more of its subguarantees have to be changed. Again, if this affects existing implementations these must be checked against their contracts and potentially adapted.
- When an existing component implementation gets changed, it needs to be rechecked against the contract(s) of that component. If the check fails, these have to be adapted (possibly leading to further changes, see previous point) or new ones introduced. The alternative will be to consider performing the change to the implementation in a different way.

Should assumptions be present, repercussions of changes are more complex because of the contravariance of refined assumptions, and more proof obligations have to be considered (see above).

2.2 STR_UT—Structured _utomata language

Hierarchical automata concepts, such as Statecharts, UML state machines etc, are in widespread use for specification because they facilitate information hiding and are more understandable. Grouping states to superstates, however, still has the disadvantage of making the structure rather rigid: as soon as superstates have been defined and refined it is not possible to dissolve them anymore, or to group them differently. Harel [27] therefore considered introducing overlapping states but has not been able to finish the concept because it resulted being too difficult [28].

Modelling automata with primitives (states, transitions/events/actions) obscures their meaning. It is difficult, if not, without hierarchy, impossible to tell apart from an automaton those features that were essential in constructing it from those that are merely incidental. Automata with more than very few states therefore typically require a lot of textual explanation to make them usable for specification. Moreover, states are not an end for specification but rather a means. It is not the states one is interested in but a specified behaviour which may be realisable in different ways, with automata containing different states and transitions. Similar to patterns for temporal logic, automata-based specifications have been combined with patterns as well [24]. The issue of rigidity of a pattern-based approach, however, remains.

We here propose a visual automata description language as an approach to specifying automata

- that on the one hand resides on a higher abstraction level than (hierarchical) automata, and
- that on the other hand is more flexible than patterns, enabling the expression of event orderings, timings and the logical relation of facts without having to care about "bubbles and arrows".

Structured automata have interfaces containing state as well as event ports. Events are, essentially, a shortcut for a certain history of states and state changes. They are also useful for the alignment with events defined by ADLs, such as “RTEEvent”. AUTOSAR events have been criticised for being hard or even impossible to locate in a system [23], but from a usability perspective they are certainly helpful. If localisability is needed one can always introduce a new state/variable associated with the event in question.

2.3 Language primitives—atomic modules

Structured automata are transfer (or stream-processing) functions, assembled from basic functions called “atomic modules”. These represent simple propositions such as (a) logical combinations of assertions, (b) state changes, event counts and event orderings, (c) state-event combinations, and (d) time intervals.

In a graphical representation they look like block diagrams that can be connected according to port types: event ports are triangular and connect to other event ports, state ports are square and connect to other state ports. Inports are white, outports are black. All modules are receptive. Composite modules are grouped as components that expose an arbitrary set of internal outports and inports (the latter must not be connected otherwise) as an interface to the environment. Following are the atomic modules defined for STR_UT so far.

Note that this is a basic STR_UT version that uses singular intervals (fixed values) for Δt timing parameters. For the purpose of implementation verification it will be useful to provide non-singular timing intervals since no real-world component can fulfil a specification like “Latency = 20µs”.

2.3.1 Propositional primitives

Propositional primitives represent Boolean functions. There are two variants: atomic modules with n inputs that specify an arbitrary Boolean function of n variables (Figure 7), annotated with the desired function expression, or atomic modules realising standard, symmetric and fixed functions such as *And*, *Or*, *Negation*, *Xor* etc (Figure 8).

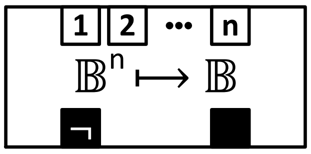
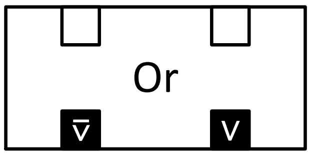


Figure 7. Generic Boolean module.

Inports. The variables of the function.

Outports. The function value (right) and the negated function value (left), corresponding to the values of the input variables at that time.



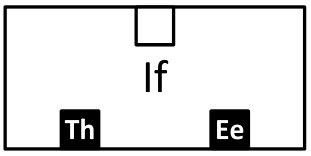


Figure 8. Disjunction (left) and equivalence/negation[5] (right).

Parameters. None.

The outport values reflect the input values or a change of input values at any time, without delay, according to the function.

2.3.2 Union

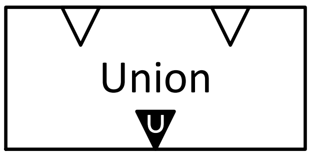


Figure 9. The *Union* module.

Inports. Two or more event ports.

Outport. One event port, marked "U".

Parameters. None.

[5] The outport marked "Th" (Then) is *true* when and only when the input is *true*, the outport marked "Ee" (Else) is *true* when and only when the input is *false*.

The *Union* module represents the union of sets of events. Whenever there is an event at one of the inports, at the same instant there is an event at the outport. If both input events happen at the same time there are two output events at that time.

2.3.3 Split

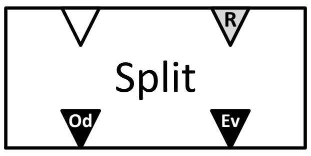


Figure 10. The *Split* module.

Inputs. One standard event port and one reset port, marked “R”.

Outputs. One event port on the right, marked “Ev” (Even) and one event port on the left, marked “Od” (Odd).

Parameters. None.

In the initial state, the first event happening at the standard inport results in a simultaneous event at the *Odd* outport. The next input event results in a simultaneous event at the *Even* outport. The third input event causes an output event at *Odd* again, the fourth, at *Even*, and so on. Odd-numbered input events (1st, 3rd, 5th, …) are reflected at the *Odd* outport, even-numbered input events (2nd, 4th, 6th, …) at the *Even* outport. An event at the reset port takes the module to the initial state.

2.3.4 Sequent

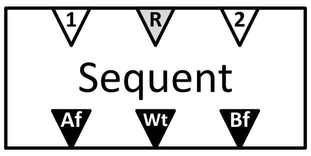


Figure 11. The *Sequent* module.

Inputs. Two standard event ports, marked “1” and “2”, and one reset port, marked “R”.

Outputs. Three event ports, marked “Af” (After), “Wt” (With) and “Bf” (Before).

Parameters. None.

From the initial state, one of the output events will take place when one of the two inports sees its first event, with the other inport having already seen at least one event, or seeing one at the same time. If both input events happen at the same instant, the outport *Wt* (“one with two”) fires an event. If inport 1 comes last, outport *Af* fires (“one after two”). If inport 2 comes last, outport *Bf* (“one before two”) fires. As soon as one of the outports fires or when the reset event takes place, the module returns to the initial state.

2.3.5 When

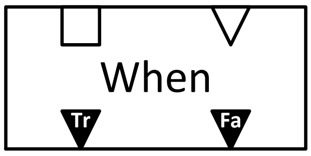


Figure 12. The *When* module.

Inputs. One state port (left), one event port (right).

Outputs. One event port marked “Tr” (True) and one event port marked “Fa” (False).

Parameters. None.

When the input event happens, the *True* outport fires if the state input is *true* at that time. If the state input is *false*, the *False* output fires instead.

2.3.6 Upon

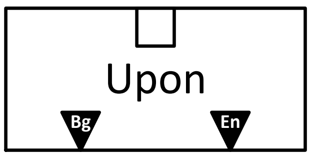


Figure 13. The *Upon* module.

Input. One state port.

Outputs. Two event ports, marked “Bg” (Beginning) and “En” (Ending).

Parameters. None.

Every time the state input becomes *true* there is an output event at *Beginning*. Every time the state input becomes *false* there is an output event at *Ending*.

2.3.7 Last

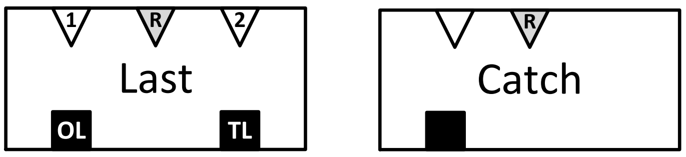


Figure 14. The *Last* module (left) and its *Catch* variant (right).

Inports (Last). Two regular event ports, marked “1” and “2”, and one reset port marked “R”.

Inports (Catch). One regular event port and one reset port marked “R”.

Outports (Last). Two state ports, marked “OL” (One last) and “TL” (Two last), respectively.

Outport (Catch). One state port.

Parameters. None.

In the initial state of Last, both outports are *false*. After that, if port 1 has seen an input event last, the *OL* outport is *true* and the *TL* outport is *false*. If port 2 has seen an input event last, the *TL* outport is *true* and the *OL* outport is *false*. The reset event takes the module back to the initial state.

The *Catch* variant switches just once. When the input event happens for the first time the outport becomes *true* and stays so until reset.

2.3.8 nTimes

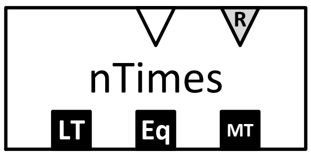


Figure 15. The *nTimes* module.

Inputs. One regular event port and one reset port marked “R”.

Outputs. Three state ports, marked “LT” (Less than), “Eq” (Equal to) and “MT” (More than).

Parameter. A whole number n > 0.

In the initial state the outport *LT* is *true* and the others are *false*. As input events keep happening and eventually their accumulated number reaches n, *LT* becomes *false*, *Eq* becomes *true* and *MT* remains *false*. When the next input event happens, taking the accumulated number of input events beyond n, *Eq* becomes *false*, *MT* becomes *true* and *LT* remains *false*. From then on, *MT* remains *true* until the reset event happens, returning the module to the initial state.

2.3.9 nFloor

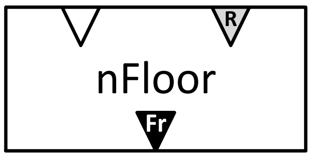


Figure 16. The *nFloor* module.

Inputs. One regular event port, one reset port marked “R”.

Outputs. One event port marked “Fr” (Floor).

Parameters. A whole number n > 1.

From the initial state and as events keep coming in, as soon as the accumulated number of input events reaches n an output event is generated. When the output event or the reset event happen the module goes back to the initial state.

This module gets its name from the floor function:

$$\left\lfloor \frac{Accumulated\ number\ of\ incoming\ events}{n} \right\rfloor$$
$$= Accumulated\ number\ of\ outgoing\ events$$

2.3.10 Δt@Most

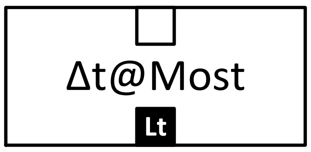


Figure 17. The *Δt@Most* module.

Inputs. One state inport.

Outputs. One state outport marked “Lt” (Late).

Parameters. Any number Δt > 0.

When, after the initial state (both ports *false*), the inport has been *true* without interruption for a time period Δt, the outport becomes *true*, indicating the expiration of the period. After that, as soon as the inport becomes *false* again, the outport switches back to *false* as well. If the inport becomes *false* before Δt expires the module returns to the initial state.

2.3.11 Δt@Least

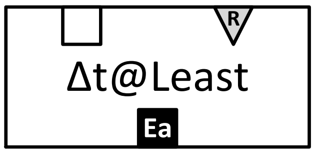


Figure 18. The *Δt@Least* module.

Inputs. One state inport, one reset port marked “R”.

Outputs. One state port marked “Ea” (Early).

Parameters. Any number Δt > 0.

When, after the initial state (both state ports *false*), the state inport becomes *true* and subsequently becomes false again without uninterruptedly having remained *true* for at least a time of length Δt, the outport becomes *true*. If the state inport remains *true* for longer than Δt the outport never becomes *true* unless a reset event occurs, taking the module back to the initial state. If the state inport is *true* when the reset event occurs the module behaves as if the state inport would have just started to be *true*.

2.3.12 ΔtDelay

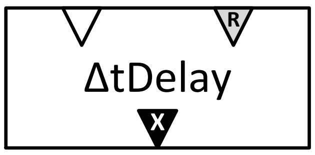


Figure 19. The *Δt@Delay* module.

Inputs. One regular event port, one reset port marked “R”.

Outputs. One event port marked “X”.

Parameters. Any number Δt > 0.

In the initial state, when an event occurs at the regular inport at time t, the outport will produce an event at time t + Δt. If, between t and t + Δt, a reset event occurs, the outport will not produce an event corresponding to the event that happened at time t, so the input event is “forgotten”. Events that follow the first event after the initial state, the firing of the outport or of the reset port with a distance of less than Δt are discarded as well.

2.3.13 ΔtCountdown

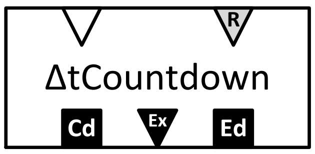


Figure 20. The *ΔtCountdown* module.

Inputs. One regular event port, one reset port marked “R”.

Outputs. One state port marked “Cd” (Counting down), one event port marked “Ex” (Expiring), one state port marked “Ed” (Expired).

Parameters. Any number Δt > 0.

In the initial state both state outports are *false*. When after that an event happens at the regular inport at time t, the *Cd* outport switches to *true* and remains *true* until time t + Δt, unless there is a reset event in the meantime. A reset returns the module to the initial state. If there is no reset, the *Cd* outport becomes *false* at t + Δt, the *Ed* outport becomes *true* simultaneously, and also the *Ex* outport fires at that time.

2.4 The elevator, revisited

Figure 21 depicts the STR_UT analogue of the elevator requirement of Section 1.3.3. The *Last* block defines, at its *OL* outport, the interval between a call and the opening of the elevator door. The *Tr* outport of the *When* module produces an event every time the *atFloor* event happens at the component inport.

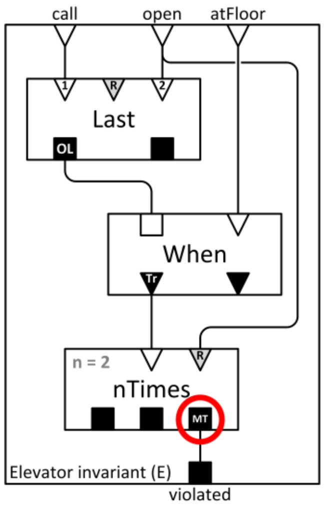


Figure 21. Elevator requirement as a STR_UT invariant.

The *nTimes* counter module is parameterised with n=2, counting the number of times the *When* block passes it an event. Whenever that event (meaning “The elevator is approaching this floor during the time between a ‘call’ and an ‘open’ event.”) has happened three or more times, a violation of the requirement is indicated. Note that for reasons of conciseness additional blocks that would keep the *Violated* outport *true* forever as soon as it has become *true* for the first time have been left out.

2.5 A simple component model

STR_UT is embedded in a component model which enables the association of invariants and contracts to software components throughout the development process, supporting a hierarchy of verified components and component types related through continuous refinement verifications (Figure 22).

Component types represent abstract specifications of components and systems and do not possess implementations. The declarative specification of a component type consists of contracts which have invariants as their assumptions and guarantees. An invariant can be a “full” invariant, meaning that it characterises a relationship between traces at component in- and outports. Preconditions relate solely to inports, postconditions accordingly are affected exclusively by outports. Contract assumptions can only be preconditions, while postconditions and full invariants serve as contract guarantees. Invariants are functions of traces of timed proposition values corresponding to the proposition being *true* or *false* of the value of its underlying trace at that time. Ports can reference multiple propositions pertaining to their trace, and they are defined in the interface of the component type they belong to.

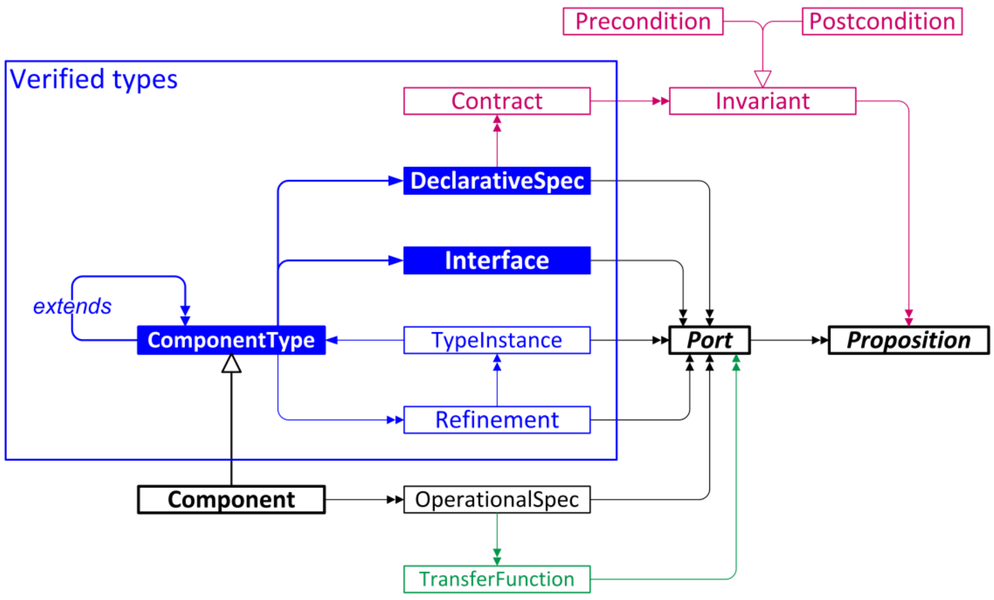


Figure 22. Meta model of verified component types.

Besides interface and declarative specification, component types may, but are not required to, have refinements. Refinements consist of instantiations of other component types and are verified with respect to their subsuming component type, following refinement verification. A component type can extend other component types, which means that the declarative specification(s) of the extended component type(s) entail(s) the declarative specification of the extending type in the usual contra-/covariant fashion.

Component types that have been sufficiently refined are able to serve as supertype of a concrete component. Concrete components have operational specifications (implementations), which can be models used for code generation, program code, binaries, and also structured automata. Operational specifications consist of transfer functions mapping the input traces at the required interface of the component to its output traces at the provided interface (see also Section 2.1):

$${}^{X}_{Req}Interface = \bigcup_{j \in J} {}^{X}Inport_j$$

$${}^{X}_{Prov}Interface = \bigcup_{k \in K} {}^{X}Outport_k$$

$$Domain\left({}^{X}Inport_j\right) \overset{\text{def}}{=} {}^{X}DI_j,\;\; Domain\left({}^{X}Outport_k\right) \overset{\text{def}}{=} {}^{X}DO_k$$

$${}^{X}Transfer : \bigcup_{j \in J}\left[{}^{X}DI_j\right]^{|T|} \mapsto \left[{}^{X}DO_k\right]^{|T|}.$$

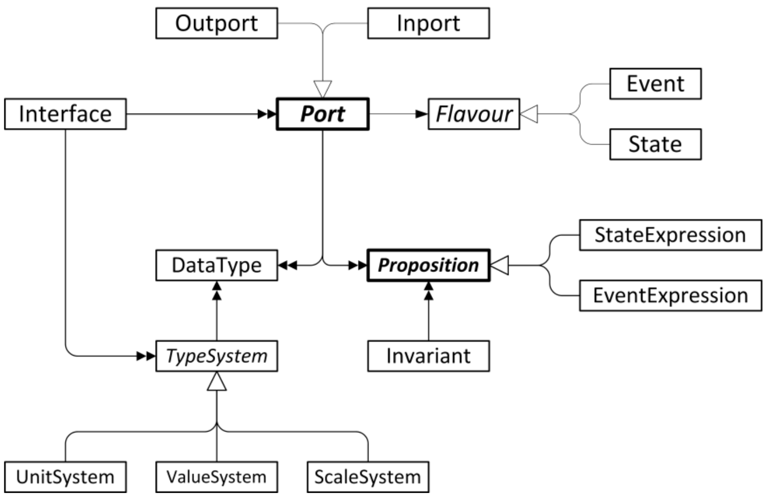


Figure 23. Ports, propositions and data types.

Figure 23 shows that ports can be in- or outports and come in an event and a state flavour. The corresponding propositions are state and event expressions (see also Figure 6).

Ports specify data types on their traces, which are rooted in the interface's type systems. Type systems can be unit systems (for example, AUTOSAR's SI-based physical unit system, realising the original idea of [59]), value systems and further type systems useful for type checking and static port compatibility checks.[6]

2.6 Structured automata grammar

The grammar of structured automata is a quadruple G = (V, Σ, R, S) of a set of nonterminal symbols *V*, a set of terminal symbols *Σ*, a set of production rules *R* and a set of start symbols *S*. Expressed in EBNF:

```
V = { <func>, <s-func>, <e-func>, <period>,
      <timepoint>, <timespan>, <nat-num> },
Σ = { 𝔹 ∪ {ε} ∪ ℝ₀⁺ },
R = { R1, R2, R3, R4, R5, R6, R7, R8, R9 },
S = { <func> }.

R1 : <func> = <s-func> | <e-func>

R2 : <s-func> =
     And(<s-func>, <s-func>, … , <s-func>) |
     Or(<s-func>, <s-func>, … , <s-func>) |
     Then(<s-func>) |
     Else(<s-func>) |
     OneLast(<e-func>, <e-func>, <e-func>) |
     TwoLast(<e-func>, <e-func>, <e-func>) |
     LessThan(<e-func>, <nat-num>) |
     EqualTo(<e-func>, <nat-num>) |
     MoreThan(<e-func>, <nat-num>) |
     Late(<s-func>, <timespan>) |
     Early(<s-func>, <e-func>, <timespan>) |
     CountingDown(<e-func>, <e-func>, <timespan>) |
     Expired(<e-func>, <e-func>, <timespan>)

R3 : <s-func> =
     true(<period>, <period>, … , <period>)

R4 : <e-func> =
     True(<s-func>, <e-func>) |
     False(<s-func>, <e-func>) |
     United(<e-func>, <e-func>) |
```

[6] We do not further consider type systems at this point. The Mars Climate Orbiter incident [66], however, exemplifies their importance.

```
          Beginning(<s-func>) |
          Ending(<s-func>) |
          OneAfterTwo(<e-func>, <e-func>, <e-func>) |
          OneBeforeTwo(<e-func>, <e-func>, <e-func>) |
          OneWithTwo(<e-func>, <e-func>, <e-func>) |
          Floor(<e-func>, <e-func>, <nat-num>) |
          Odd(<e-func>, <e-func>) |
          Even(<e-func>, <e-func>) |
          Delay(<e-func>, <e-func>, <timespan>) |
          Expiring(<e-func>, <e-func>, <timespan>)

     R5 : <e-func> =
          ε | true(<timepoint>, <timepoint>, … ,
               <timepoint>)

     R6 : <period> = [<timepoint>, <timepoint>)

     R7 : <timepoint> = T ∈ ℝ₀⁺

     R8 : <timespan> = T ∈ ℝ₀⁺

     R9 : <nat-num> = N ∈ ℕ
```

Rules 1, 2 and 4 build the structured automaton and determine its meaning. Rules 3, 5 and 9 are used to assign concrete values to the arguments of a function (alternatively, nonterminals may remain as free variables to be published at a component interface). In the case of a state-like argument, rule 3 determines the periods during which the argument is true. In the case of an event-like argument, rule 5 assigns the impossible event ε ("never") or those points in time where the event does happen. Rule 6 defines periods as left-closed and right-open (continuous) time intervals. Rules 7 and 8 determine the (absolute) values of points in time and the (relative) values of the length of timespans.

This grammar must be regarded as preliminary since STR_UT primitives can represent multiple functions. For example, the *Upon* module subsumes the functions *Beginning*(<s-func>) and *Ending*(<s-func>). Therefore the grammar remains to be adapted to the graphical nature of structured automata (possibly as a graph grammar). We postpone discussion of the mapping of functions to state and event expressions of the ports of complex modules, as well as the formal description of the instantiation of STR_UT components and of their interconnection.

2.7 Expressing TADL constraints

Figure 24 shows the representation of the TADL constraints "Simple ordered synchronisation constraint" and "Repetition constraint". The first constraint requires that input event 1 should happen before input event 2 and that the interval between the events must not exceed a given time period Δt. The STR_UT invariant captures violations repeatedly.

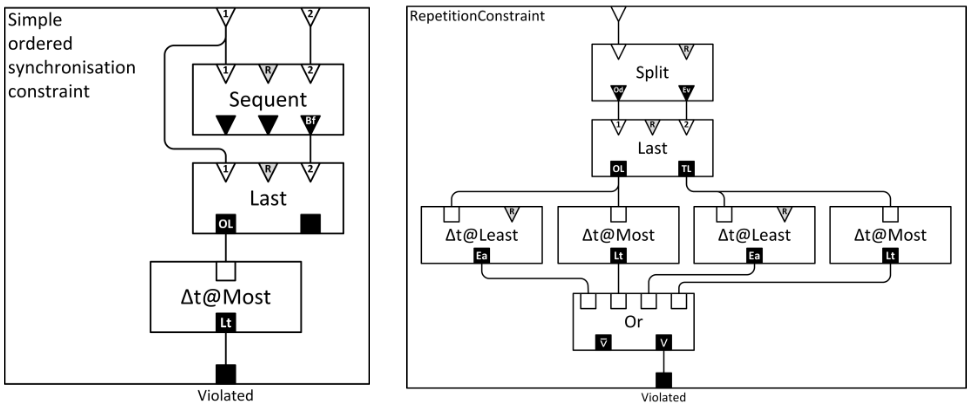


Figure 24. Two TADL timing constraints as STR_UT modules.

For the second constraint the incoming event stream is split in two alternating event streams and checked for minimal and maximal temporal distance between successive events (for a method of direct translation of EAST-ADL timing constraints to TA, see [61]). Again, for reasons of conciseness, but also because it may be useful in quantitative analyses, violations are not made permanent.

2.8 Specification bootstrapping

Structured automata are self contained in the sense that STR_UT components which are used as declarative specifications for other components become themselves modelled as transfer functions that do not have (nor usually need) a declarative specification. Currently, STR_UT transfer functions produce Boolean traces only. This does not, in principle, affect their expressiveness, but an extension towards arbitrary data types would make it easier to model transfer functions producing arbitrary discrete values and thus general automata.

3 Case study

The Stop/start system of this case study is similar to those present in many modern cars: depending on pedal action, gear status, vehicle speed etc, the motor is automatically stopped and started. There are safety as well as functional requirements the system has to satisfy.[7] Three of them we will try to model and decompose, proving that fulfilment of the decomposition is sufficient for the original requirement to hold. This should demonstrate the application of refinement verification.

3.1 Components of the Stop/start system

The part of the Stop/start system that was selected for the case study comprises eight components whose main purpose and input/output variables are stated in Table 2. See also Figure 40 and Figure 41. Some variables not considered in this case study were removed ("*Purged*") from the model, others have been given generic names ("StrtInhbt_A", for example).

Component name	**Source >> Input**	**Output >> Destination**
	Component description	
Start inhibition assessment	Env >> (StrtInhbt_A) Env >> (StrtInhbt_B) Env >> (StrtInhbt_C)	StartInhbt >> StrgyStateCoord | UsrIndcdStart | SysIndcdStart
	Whenever there is a start inhibitor active at the input, start inhibition is signalled at the output.	
Stop inhibition assessment	Env >> (StopInhbt_X)	StopInhbt >> UsrIndcdStop
	(Pass-through component in this version; no effect.)	
User-induced start	Env >> EngState Env >> GearNtrl	StartReqUsr >> EngModeCtrl

[7] For a detailed description of a similar Stop/start system, refer to [57].

	Env >> CpedState StrgyStateCoord >> StopStartStrgyState StartInhbtAsmnt >> StartInhbt	(*Purged*) >> SysIndcdStart
	Produces a user-induced start request, depending on the engine state (off/running), the state of the Stop/start system (active/not in operation), the absence of a start inhibitor, the position of the clutch pedal and whether the gear is in neutral.	
System-induced start	Env >> (*Purged*) Env >> (*Purged*) Env >> (*Purged*) Env >> EngState Env >> GearNtrl StrgyStateCoord >> StopStartStrgyState StartInhbtAsmnt >> StartInhbt UsrIndcdStart >> (*Purged*)	StartReqSys >> EngModeCtrl
	Produces a system-induced start request, depending on the engine state, the state of the Stop/start system, the absence of a start inhibitor, and other variables. A system-induced start may be triggered if the speed of the vehicle is too high when the motor gets stopped, for example.	
User-induced stop	InhbtLogic >> StopInhbt StrgyStateCoord >> StopStartStrgyState Env >> EngState Env >> CpedState Env >> TrnGearNtrl	StopReqUsr >> EngModeCtrl
	Produces a user-induced stop request, depending on the engine state, the state of the Stop/start system, the absence of a stop inhibitor, the state of the clutch pedal and the state of the gearbox. A user-induced stop may be triggered if the gear is in neutral and the clutch pedal is being released.	
Strategy state coordination	StartInhbtAsmnt >> StartInhbt Env >> EngCtlCrsh Env >> EngState	StopStartStrgyState >> UsrIndcdStart | SysIndcdStart | UsrIndcdStop |

		EngModeCtrl
	Sets the state of the Stop/start system. For example, if the motor is stopped and a start inhibitor is present, or if a crash is indicated, the system goes out of operation.	
Engine mode control	StrgyStateCoord >> StopStartStrgyState Env >> EngState UsrIndcdStop >> StopReqUsr UsrIndcdStart >> StartReqUsr SysIndcdStart >> StartReqSys	EngModeCmd >> OutFsn
	Produces a stop or start request if the Stop/start system is active and there is a user- or system-induced stop or start request.	
Output fusion	EngModeCtrl >> EngModeCmd Env >> (*Purged*) Env >> StopStartMode	EngCmd >> Env
	Sends out a stop or start request to engine control if it gets a corresponding request from engine mode control, provided the Stop/start system is in normal mode and engine control is not determined to be faulty.	

Table 2. Components of the Stop/start system with their in- and output variables.

For the refinement proofs of three verification conditions multivalued variables were reduced to pseudo-Boolean according to Table 3.

Variable	Value range
CpedState	**RELEASED,** PRESSED, **DEPRESSED**
EngCmd, EngModeCmd	**RUN, STOP,** DEACTIVATED
EngCtlSfty	**OK, FAULTY**
EngState	**RUNNING, STOPPED,** IGNORE
StopStartMode	**NORMAL,** STARTUP, ERROR
StopStartStrgyState	**ACTIVE, DEACTIVATED (= *NOT_IN_OPERATION*)**

Table 3. Value ranges of Stop/start system variables.

Bold values indicate those variable values that were retained. All other variables were Boolean from the start (see Table 2). To further simplify modelling, the variables *StopStartMode* and *StopStartStrgyState* were merged into a single variable *SSSMode*, with the values *ACTIVE* and *DEACTIVATED*.

3.2 Verification conditions—specification & analysis

Verification conditions were simplified in a faithful way. Figure 27, Figure 32 and Figure 35 show the original requirements. Since they are unconditional we were able to use the simple proof schema (Section 2.1) to show refinement:

$$[G_1 \wedge G_2 \wedge \ldots \wedge G_n] \rightarrow G.$$

Expressed verbally, this means that the conjunction of (unconditional) subcomponent guarantees implies the (unconditional) system guarantee. Refinements were created heuristically by consideration.

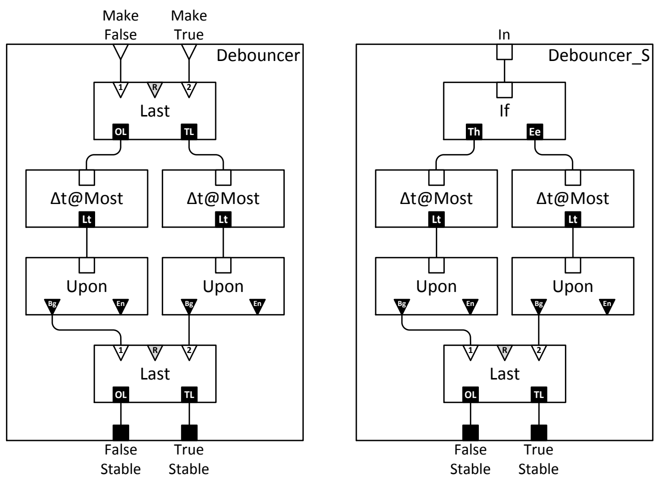


Figure 25. Debouncer and Debouncer_S templates.

Timing was modelled by dedicated debouncer modules or similar constructions (Figure 25).

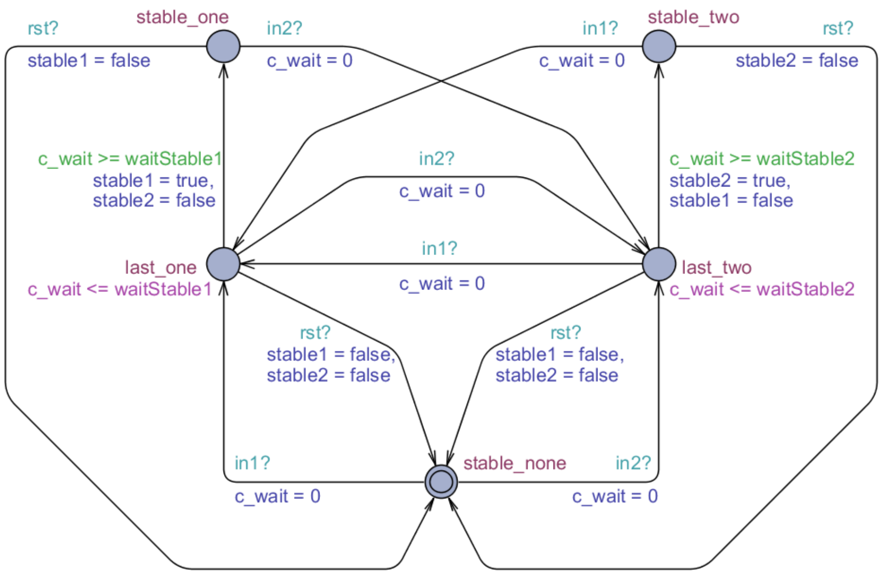


Figure 26. Debouncer template, translation to UPPAAL as a single TA.

Figure 26 depicts the timed automaton corresponding to the Debouncer module with event inports (Figure 25, left), when it is translated as a whole instead of as a network of six separate TA. To reduce the numbers of automata in translations, which seems to create problems for UPPAAL due to the greater number of interleavings, we merged several TA into one at several points. Even though this is a simplification it was done in a way that should not have affected analysis results.

3.2.1 Do not start with engaged power train

(1) FT_Safety_NoCommandedStartWithPowertrainEngaged
Do not command a start if gear not neutral and clutch not depressed.
cyclic_P_implies_Q_at_step_X_thereafter__immediate [max_X = 1, **PA1**]

```
[P] ( StopStartMode == STOP_START_MODE_NORMAL )  &&
    ( !TrnGearNtrl )    &&
    ( CpedState != CLU_DEPRESSED )
[Q] !tr( EngCmd == ENG_CMD_RUN )
```

Figure 27. Verification condition 1.

Specification. Figure 28 shows the STR_UT module representing safety requirement 1. It was refined into four component contracts for *User-Induced Start*, *System-Induced Start* (Figure 29), *Engine Mode Control* and *Output Fusion* (Figure 30).

The guarantee of *User-Induced Start* states that the component must not produce a start command when the clutch is not in a stable (read: debounced) state or the gear is not in neutral. Note that throughout the case study verification conditions are such that any violation of a guarantee is permanent.

The guarantee of *System-Induced Start* states that the component must not produce a start command when the gear is not in neutral position.

Engine Mode Control passes on incoming start requests, thus its guarantee ensures that it can only do so in a certain time window following an incoming request.

Likewise, *Output Fusion* has a time window available to forward incoming start requests. At the same time, it has to ensure that the Stop/start system is active. Otherwise it must remain silent.

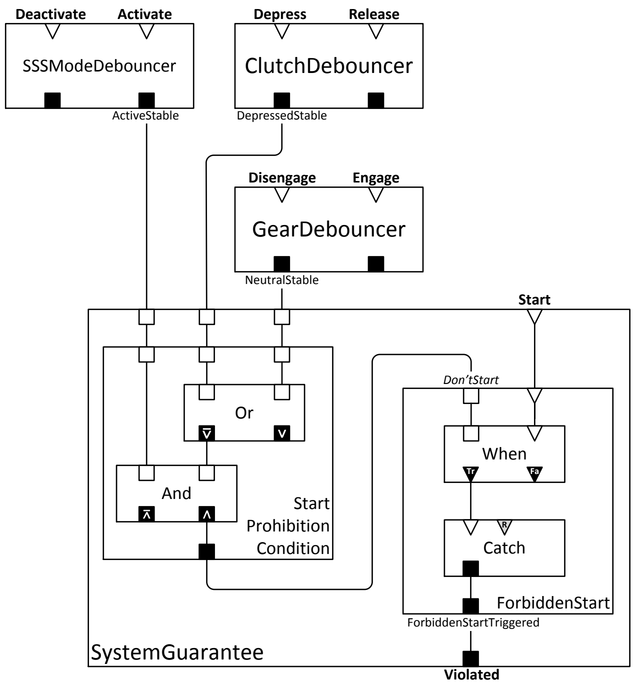


Figure 28. System guarantee, verification condition 1.

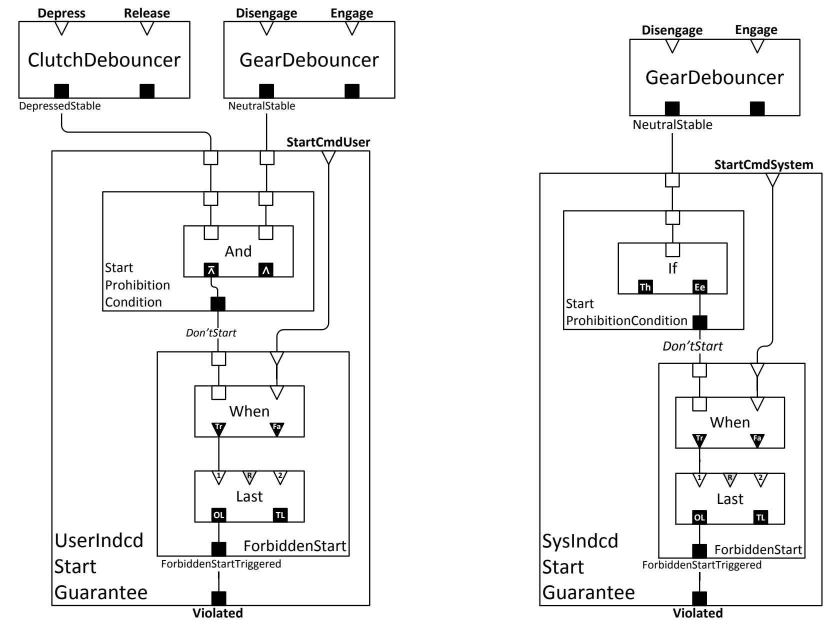


Figure 29. Guarantee of components *User-Induced Start* and *System-Induced Start*.

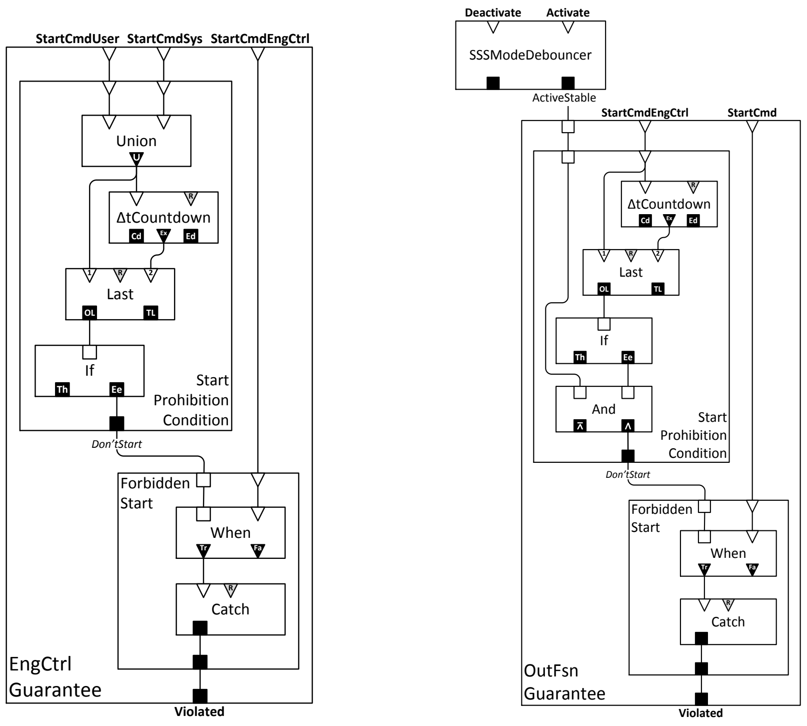


Figure 30. Guarantees of components *Engine Mode Control* and *Output Fusion*.

Analysis. All STR_UT guarantees were automatically translated to a network of TA and UPPAAL analysis started. Because UPPAAL did not return even after several hours we concluded that the state space created by about 30 TA was too large to handle for the model checker. After some manual equivalence transformations (Figure 26 shows the debouncer as an example) to reduce the number of TA in the network, analysis was successful.

The settings of time parameters that enabled verification demonstrated the interaction of system structure and timing: the debouncing period of the debouncers of the system guarantee had to be greater than the sum of the time window allowances of the *Engine Mode Control* and *Output Fusion* guarantees, since these are connected "in series" by the data flow (see Figure 40 and Figure 41), making their latencies add up.

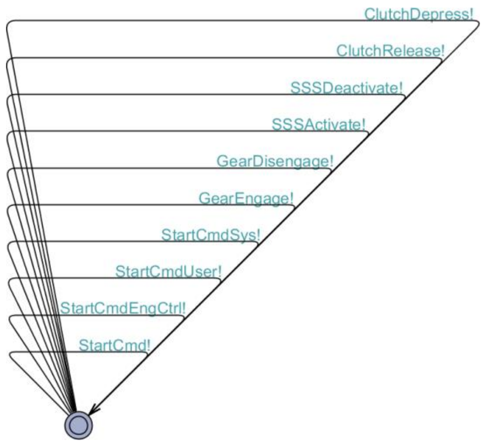
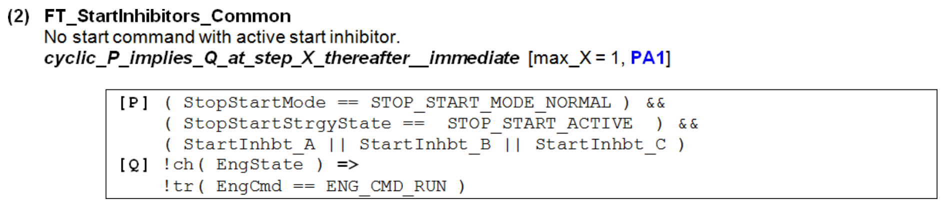


Figure 31. Environment model for UPPAAL analysis of Stop/start system requirements.

The analysed TA network eventually comprised eleven automata with nine clocks altogether. The environment was modelled by a twelfth automaton triggering any input event anytime (Figure 31). To ensure that a violation of the system guarantee, to be counted as a failure, must occur strictly before the violation of any component guarantee, an additional clock for the system guarantee had to be introduced. The vacuity check took 22 seconds, and the refinement proof took 219 seconds.

This and all other analyses were performed on an Intel i7 machine with 3.2 GHz and 8 GB RAM, running the 64-bit UPPAAL version 4.1 on Linux.

3.2.2 Do not start while a start inhibitor is present



(2) **FT_StartInhibitors_Common**
No start command with active start inhibitor.
cyclic_P_implies_Q_at_step_X_thereafter__immediate [max_X = 1, **PA1**]

```
[P] ( StopStartMode == STOP_START_MODE_NORMAL ) &&
    ( StopStartStrgyState ==  STOP_START_ACTIVE  ) &&
    ( StartInhbt_A || StartInhbt_B || StartInhbt_C )
[Q] !ch( EngState ) =>
    !tr( EngCmd == ENG_CMD_RUN )
```

Figure 32. Verification condition 2.

Specification. Figure 33 shows the STR_UT module for safety requirement 2. It was refined into two component contracts for *Strategy State Coordination* and *Engine Mode Control* (Figure 34). Note that the *If* module in the Start prohibition condition of the system guarantee is just a placeholder making the state of the outport of the Start prohibition condition identical to the inport state.

The system guarantee states that there must not be a start request when a start inhibitor is present.

The guarantee of *Strategy State Coordination* has two parts: (1) in the moment a start inhibitor appears, the system must either be already deactivated, or the system must deactivate within a specified timeframe; (2) as long as there is a start inhibitor present, the system must not leave the deactivated state.

The guarantee of *Engine Mode Control* states that the component shall not issue a start request unless the Stop/start system is active.

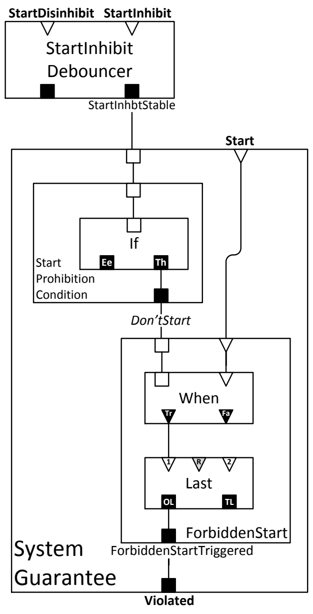


Figure 33. System guarantee, verification condition 2.

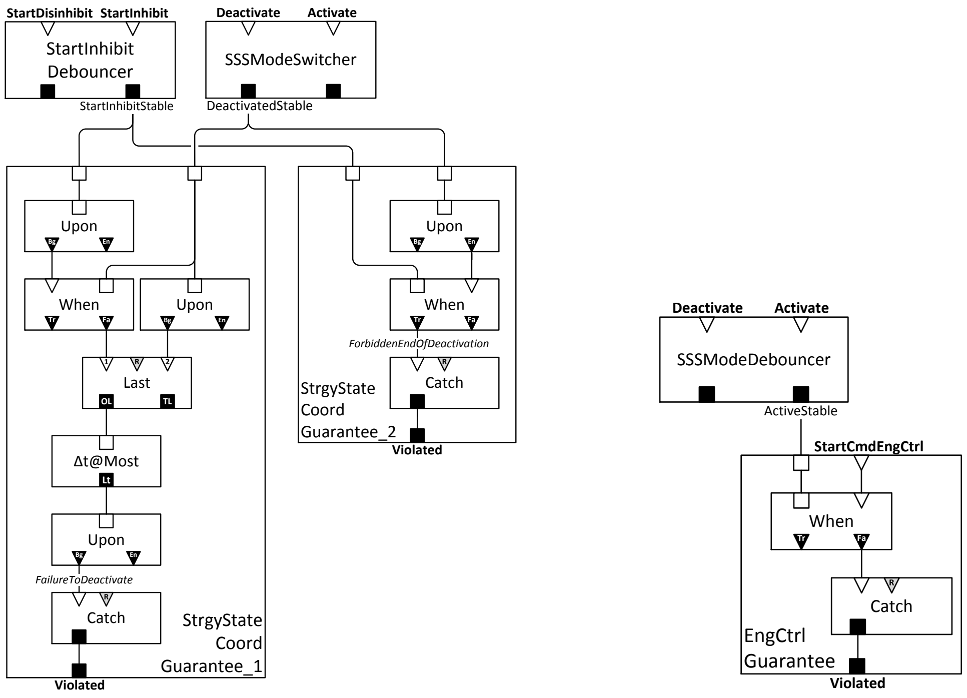


Figure 34. Guarantees of components *Strategy State Coordination* and *Engine Mode Control*.

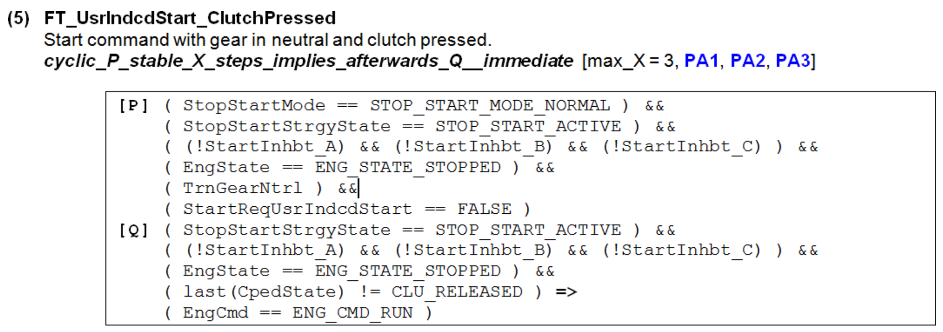
Analysis. The analysed TA network was not manually simplified except for the debouncers, and it comprised 19 automata containing five clocks, including, as before, a clock for the system guarantee. To make sure that the system guarantee cannot be violated before the subcomponent guarantees, timings play a vital role:

- the delay of the Start inhibit debouncer of the system guarantee (represented by the Δt parameter of the left *Δt@Most* module of the debouncer) regarding an incoming Start disinhibit event must be shorter than the corresponding delay in the Strategy state coordination guarantee;
- the delay of the Start inhibit debouncer of the system guarantee regarding an incoming Start inhibit event must be longer than the sum of the corresponding delay and the *Δt@Most* time window length in the Strategy state coordination guarantee, and the delay of the SSS mode debouncer in the Engine mode control guarantee.

The timing values chosen for UPPAAL analysis were: 2 and 10 for the Start inhibit debouncer of the system guarantee, 3 for all other timings (debouncers and the single *Δt@Most* of Strategy state coordination guarantee 1).

For verification condition 2, each environment input event was automatically translated to a separate automaton (with one state and one transition, for an additional six TA on top of the 19). Both the vacuity check and the refinement proof took no more than one second. See Appendix B for a detailed presentation of this UPPAAL model.

3.2.3 Command a start when clutch is being pressed while gear is disengaged



(5) **FT_UsrIndcdStart_ClutchPressed**
Start command with gear in neutral and clutch pressed.
cyclic_P_stable_X_steps_implies_afterwards_Q__immediate [max_X = 3, **PA1**, **PA2**, **PA3**]

```
[P] ( StopStartMode == STOP_START_MODE_NORMAL ) &&
    ( StopStartStrgyState == STOP_START_ACTIVE ) &&
    ( (!StartInhbt_A) && (!StartInhbt_B) && (!StartInhbt_C) ) &&
    ( EngState == ENG_STATE_STOPPED ) &&
    ( TrnGearNtrl ) &&
    ( StartReqUsrIndcdStart == FALSE )
[Q] ( StopStartStrgyState == STOP_START_ACTIVE ) &&
    ( (!StartInhbt_A) && (!StartInhbt_B) && (!StartInhbt_C) ) &&
    ( EngState == ENG_STATE_STOPPED ) &&
    ( last(CpedState) != CLU_RELEASED ) =>
    ( EngCmd == ENG_CMD_RUN )
```

Figure 35. Verification condition 5.

Specification. Figure 36 shows the STR_UT module representing requirement 5, which is a functional rather than a safety requirement. It was refined into two component contracts, for *User-Induced Start* and *Engine Mode Control*, respectively. The system guarantee states that the motor shall be started when it is presently stopped, the Stop/start system is active, the gear is disengaged, there is no start inhibitor, and the clutch is being pressed. The *Debouncer_S* module ensures that the motor, SSS mode, gear and start inhibitor preconditions must simultaneously be fulfilled for a specified timespan before the clutch action can trigger a start.

The guarantee of *User-Induced Start* asserts that the motor is started when it is presently stopped, there is no start inhibitor, the gear is disengaged and the clutch is being pressed (that is, the *ReleasedStable* state is ending).

The guarantee of *Engine Mode Control* states that a start request will be issued within a certain timeframe whenever there has just been an incoming start request (from the user) and the system is active (Figure 37).

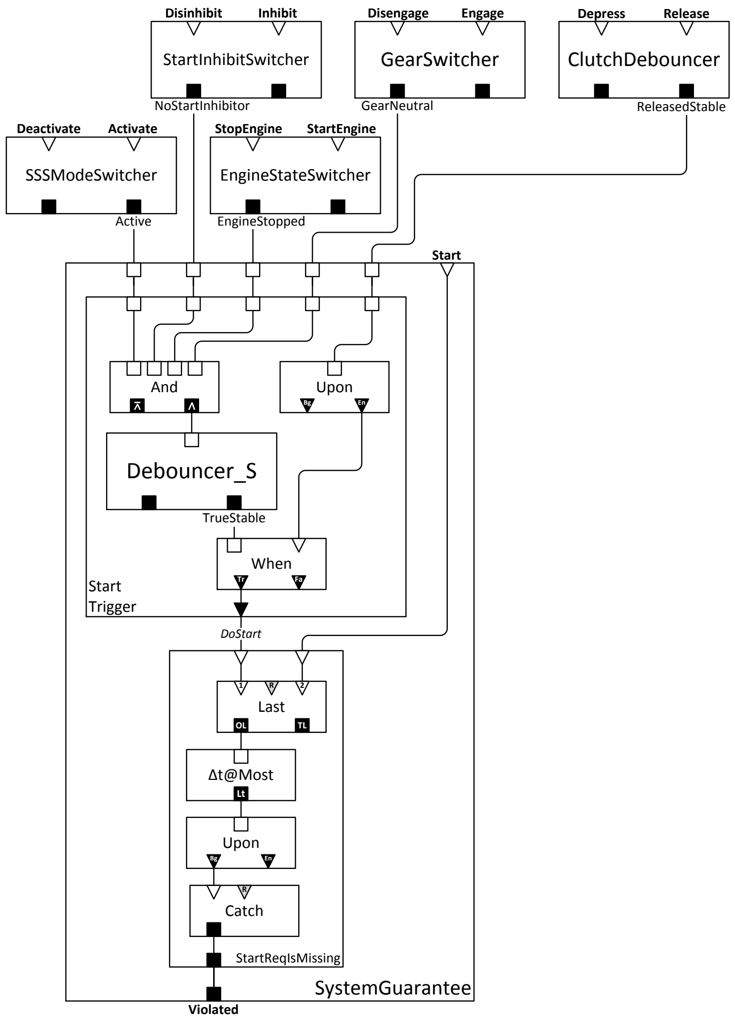


Figure 36. System guarantee, verification condition 5.

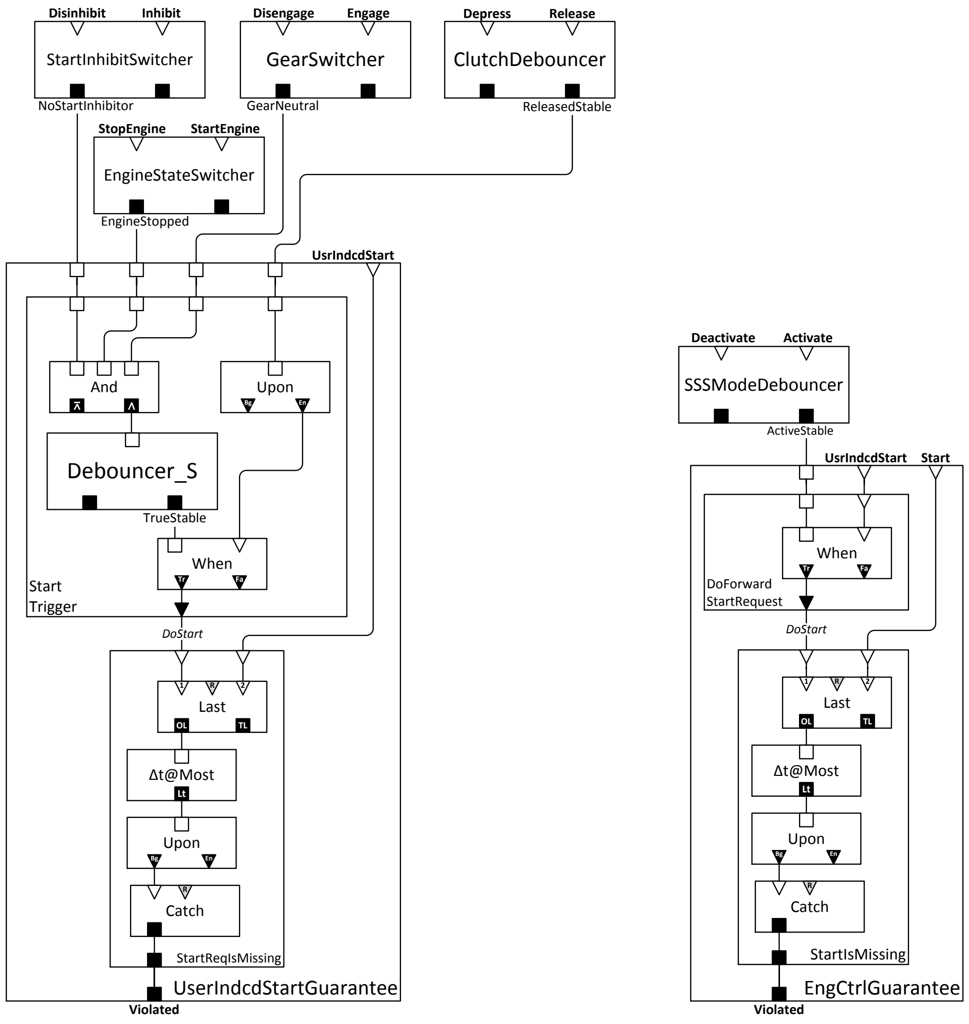


Figure 37. Guarantees of components *User-Induced Start* and *Engine Mode Control*.

Analysis. The (manually simplified) UPPAAL network to analyse comprised 17 TA with eight clocks altogether. Timings that were tried in combination lay between 2 and 50 units; we have not looked for tightness there yet. The vacuity check finished in seven seconds while refinement verification took 180 seconds with the same computer configuration as before (Section 3.2.1).

3.3 Summary

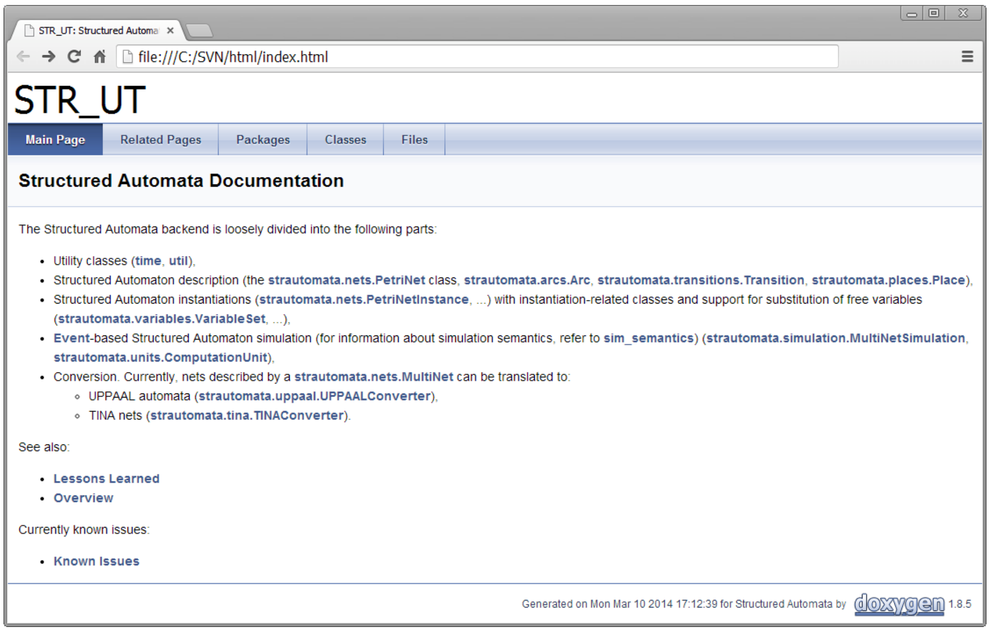
All three verification conditions chosen were, separately, able to be refinement verified. Before they can be verified together, analysis efficiency must be improved, particularly as timeless transitions interleaving is concerned.

Some analysis-related observations:

- UPPAAL may not cope well with networks of many TA, so we merged automata by hand to perform analysis of the refinement of verification conditions 1 and 5. The reason could be the number of transition interleavings that a large TA network produces.
- The influence of data flow structure on timing was demonstrated.
- An analysis with TINA should be tried side by side with UPPAAL. We expect that the Petri net-based TINA toolbox could, in certain aspects, be better suited for the modular STR_UT approach. Translations are simpler (see Section 4.2), and no intervention to build a product automaton and to simplify it manually will be necessary.
- The rather large state spaces of the UPPAAL models of verification conditions 1 and 5 have revealed the need to look into the interleaving issue. We expect to improve performance by adapting the way transition priorities are handled.

Regarding the usability of the STR_UT notation, all properties encountered could be expressed quite straightforwardly, from event counting (Section 2.4) and TADL constraints (Section 2.7) to debouncer modules (Section 3.2). Understandability-wise we feel it to be close to the PSL/SALT approach (Section 1.3.2), while the visuality of STR_UT and its similarity to well-known block diagram notations in engineering might give it an advantage for use in graphical modelling environments. This impression is, of course, biased and will have to be substantiated by further investigation.

4 Tool support



STR_UT

Structured Automata Documentation

The Structured Automata backend is loosely divided into the following parts:

- Utility classes (**time**, **util**),
- Structured Automaton description (the **strautomata.nets.PetriNet** class, **strautomata.arcs.Arc**, **strautomata.transitions.Transition**, **strautomata.places.Place**),
- Structured Automaton instantiations (**strautomata.nets.PetriNetInstance**, ...) with instantiation-related classes and support for substitution of free variables (**strautomata.variables.VariableSet**, ...),
- **Event**-based Structured Automaton simulation (for information about simulation semantics, refer to **sim_semantics**) (**strautomata.simulation.MultiNetSimulation**, **strautomata.units.ComputationUnit**),
- Conversion. Currently, nets described by a **strautomata.nets.MultiNet** can be translated to:
  - UPPAAL automata (**strautomata.uppaal.UPPAALConverter**),
  - TINA nets (**strautomata.tina.TINAConverter**).

See also:

- **Lessons Learned**
- **Overview**

Currently known issues:

- **Known Issues**

Generated on Mon Mar 10 2014 17:12:39 for Structured Automata by doxygen 1.8.5

Figure 38. Doxygen documentation of the STR_UT backend (screenshot).

In the course of the project a library was developed for the automatic transformation of STR_UT modules and components to (1) networks of UPPAAL Timed automata, and (2) Time Petri nets in Tina format.

4.1 Translation—STR_UT to UPPAAL

There is no obvious way to translate arbitrary STR_UT modules to UPPAAL automata automatically but one can transfer the semantics of a fixed STR_UT to an UPPAAL automaton as long as certain conditions are met (for instance, UPPAAL automata have problems with some kinds of passive delays). Usually, structured automata are constructed from a set of base modules like *And*, *When* or *ΔtCountdown* modules, which are instantiated and interconnected by ports to form a STR_UT instance. These modules can be translated to UPPAAL format manually. Any STR_UT instance that is a composition containing these basic modules only can then be translated to UPPAAL by properly instantiating and connecting the translations of the basic modules it consists of.

Each STR_UT module has a set of incoming and outgoing ports which may be either event or state ports. Additionally, some basic modules have free parameters, for example the latency of a *ΔtDelay* module; the value that is assigned to these parameters can differ for different instantiations of the same module. Just like STR_UT, networks of UPPAAL automata consist of possibly synchronised instances of UPPAAL templates. In a translation of a STR_UT template to an UPPAAL automaton, each STR_UT atomic module therefore corresponds to an UPPAAL template and each instance of an atomic module corresponds to an instance of the respective UPPAAL template.

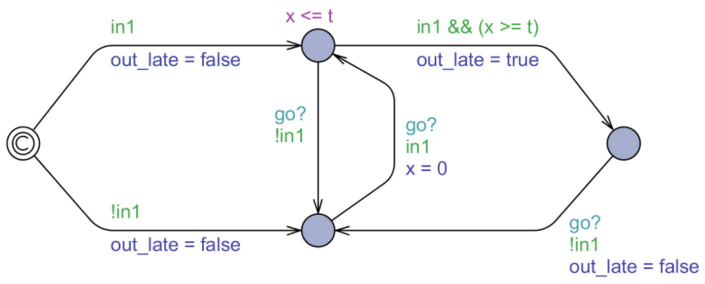


Figure 39. ***Dt@Most*** **module as UPPAAL automaton.**

Some modules may need to be forced to take a transition whenever possible, for example, when one of their state inports changes value. This is realised by adding an additional template called “Go” to the resulting UPPAAL automaton. The *Go* template tries to synchronise on a global urgent channel called “go_chan!” (similar to [68]). Each enforced transition synchronises on “go_chan?” and thus must be taken if possible (Figure 39). The *Go* template is instantiated exactly once in every UPPAAL automaton generated by the translation process and does not correspond to any module on the STR_UT side.

UPPAAL automata do not have a port concept. Instead, the templates can have parameters that take arguments by value or by reference. The concept of ports is realised with reference parameters. State ports are realised by passing a reference to a Boolean variable that can be changed by the providing side of a state-state port connection. Note that the input module/automaton must be able to cope with changing values for its input variable at any time (receptivity). In particular it cannot prevent the input variable from changing in any way. Event ports are realised by passing a reference to a channel. With unqualified UPPAAL channels, as opposed to reference variables, to synchronize on *channel!* requires at least one module trying to synchronize on *channel?*, otherwise the transition cannot be taken. Unlike the situation in STR_UT an event inport could prevent an event outport from emitting an event if unqualified UPPAAL channels were used. Therefore, an event-event port connection is realised by passing a reference to a broadcast channel b to the instances of the involved templates. The providing side then synchronizes on *b!* and the requiring side synchronises on *b?*. In this setup, the providing side cannot be influenced by the input side (just like it is in STR_UT).

To translate a STR_UT module containing instances of atomic modules to an UPPAAL automaton U, the following steps are performed:

(1) Begin.

(2) Add the *Go* template to U and instantiate it once.

(3) For each atomic module in S, add the corresponding template from the template library to U.

(4) For each outport instance O in S, add a variable V(O) to U. The variable is of type bool if O is a state port and of type broadcast channel if O is an event port.

(5) Associate V(O) with every inport J connected to O by setting V(J) = V(O).

(6) For each inport I that is not connected to any outport in S, add a variable V(I) to U.

(7) For every module instance M in S, do

   a. Identify the UPPAAL template T that corresponds to the atomic module M is an instance of.

   b. Identify the arguments that have to be passed to T to create an instance that matches M. For this purpose, identify which parameter of T maps to which port instance P of M and pass the corresponding port variable V(P) for that parameter. Also, extract from M the set of additional (non-port) arguments (such as delay times) that have to be passed to T. This results in a set A of arguments.

   c. Add an instance T(A) to U.

(8) End.

4.2 Translation—STR_UT to Tina

Converting Structured automata modules to Time Petri nets (TPN) in Tina [36] is straightforward, as the semantics of the Petri net representation of STR_UT modules is close to the semantics of Tina. There are, however, a few differences.

The major difference between Tina PN and STR_UTs is modularity. While the modularity of STR_UTs facilitates the construction of complex automata by module reuse it does not lead to an increase in expressive power, and thus it does not incur a greater analysis effort.

It is quite easy to flatten all instances of atomic modules contained in a STR_UT to just one single module instance. Atomic modules are put together in a puzzle-like fashion, and there is no need to generate a product automaton. The key step is this: a state-state port connection of two module instances can be represented by performing a disjoint union on the node set of the PN representation of both modules and identifying those places mapped to each other on both ends of the connection. The result is a larger Petri net that as a whole behaves like the two input nets combined. Similarly, an event-event port connection can be realised by merging the two nets and identifying the transitions mapped to the ports with each other.

When the different modules of a STR_UT have been merged into a single PN, STR_UT transition priorities remain to be translated to priorities compatible with Tina. While priorities in STR_UT are established by attaching a floating-point value to each transition, in Tina priorities are set by imposing an order on them.

5 Conclusion

This report presents the results of the joint FAT/RWTH project aimed at enhancing the semantic description of automotive (cyber-physical) software components and at facilitating the tool-supported industrial application of formal methods in CPSW design:

- We have developed a novel, visual approach to specifying automata with timing that in turn can be used to formally specify the behaviour of cyber-physical software components (possibly by means of assume-guarantee contracts).
- This approach facilitates refinement verification, a concept for the assurance of systematic, verified compatibility, exchangeability and reuse of CPSWC.
- The STR_UT language is embedded in a component model continuously supporting the CPSW development process from the level of functional design until implementation.
- The STR_UT language reduces the semantic gap between requirements in natural language and corresponding operationalised, formal requirements. STR_UT is graphical and its block diagrams resemble well-known, tool-supported software engineering notations. We expect it to further the consistent and effective application of formal methods in industrial practice.
- A library providing translations to UPPAAL TA and TINA TPN has been developed. With its help it was possible to verify, in UPPAAL, the refinement of three verification conditions (out of a total of six) selected for the Stop/start system case study.
- In the case study we were able to demonstrate the influence of application-specific timing.

Regarding use cases (Section 1.2), u/c Z has been completely, u/c A has been partially demonstrated. Use case B/C has been discussed on the background of refinement verification. Use cases D and E were not considered.

In practice it might be hard to prove refinement of every detail of every contract on every level, and the web of proven refinement relations will not be complete. This is, however, not a matter of all-or-nothing.

So far we have used unconditional guarantees and have not included contracts with assumptions. Furthermore, every verification condition has been refined and verified on its own, while in reality their refinements have to be verified together, just as a potential implementation must realise all verification conditions at the same time. Both aspects are potential threats to feasibility. TA and TPN cannot capture preemption since it would require stopwatch models to do so, and stopwatch automata are undecidable [29]. A viable method using overapproximation has been proposed [16], which might open a way to timing analysis of preemptive scheduling.

Acknowledgement

The authors would like to thank all members of the FAT working group 31 for valuable feedback during this project.

Phillip Keldenich’s contribution, as a student worker at the Embedded Software Laboratory (RWTH i11), in developing the tool support and performing model translations and analyses is much appreciated.

Abbreviations & acronyms

AADL	Architecture analysis and description language (formerly: Avionics architecture description language)
ADL	Architecture description language
CPS	Cyber-physical system
CPSW	Cyber-physical software
CPSWC	Cyber-physical software component
CTL	Computation tree logic
EBNF	Extended Backus-Naur form
LTL	Linear temporal logic
OEM	Original (automotive) equipment manufacturer
PN	Petri net
RTE	Run-time environment
STR_UT	Structured automaton/a (language)
TA	Timed automaton
TPN	Time Petri net
VFB	Virtual functional bus

Glossary

This glossary gives definitions and explanations of terms used in this report.

Component instance

The instantiation of a component as a subcomponent in the composite realisation of another component.

Composite realisation

The refinement of a component based on collaborating subcomponents.

Declarative specification

A component specification by invariants.

Invariant

A function from propositions pertaining to component ports to a Boolean trace. The value of the Boolean trace at time *t* specifies whether the invariant holds at that point.

Operational specification

A component specification by transfer functions. An "implementation" is an operational specification written in, for example, C, C++, Structured Text etc.

Postcondition

A function from propositions pertaining to component outports to a Boolean trace. Postconditions are a specialisation/subtype of invariants.

Precondition

A function from propositions pertaining to component inports to a Boolean trace. Preconditions are a specialisation/subtype of invariants.

Refinement verification

The technique of proving that a given decomposition of a higher-level specification/contract refines the latter.

Transfer function

A function from the traces of the inports of a component to the trace of an outport of that component.

Sources

[1] Abid, N, S Dal Zilio, D Le Botlan (2012). Real-time specification patterns and tools. In: Stoelinga, Pinger (eds), *FMICS 2012*. LNCS 7437: 1-15.

[2] ATESST2 consortium (2010). EAST-ADL Domain model specification, version 2.1.11. http://www.atesst.org, 2013-01-30.

[3] AUTOSAR administration (2010). Software component template, version 3.3.0. http://autosar.org/, 2013-02-11.

[4] Barnat, J, P Bauch, L Brim (2012). Checking sanity of software requirements. In: Eleftherakis, Hinchey, Holcombe (eds), *SEFM 2012*. LNCS 7504: 48-62.

[5] Bauer, A, M Leucker (2011). The theory and practice of SALT. In: Bobaru et al (eds), *NFM 2011*. LNCS 6617: 13-40.

[6] Baumgart, A, E Böde, M Büker, W Damm, G Ehmen, T Gezgin, St Henkler, H Hungar, B Josko, M Oertel, Th Peikenkamp, Ph Reinkemeier, I Stierand, R Weber (2011). Architecture modeling. Research report, OFFIS Oldenburg.

[7] Berry, G, G Gonthier (1992). The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19: 87-152.

[8] Berthomieu, B, J-P Bodeveix, C Chaudet, S Dal Zilio, M Filali, F Vernadat (2009). Formal verification of AADL specifications in the Topcased environment. In: Kordon, Kermarrec (eds), *Ada-Europe 2009*. LNCS 5570: 207-221.

[9] Berthomieu, B, J-P Bodeveix, S Dal Zilio, P Dissaux, M Filali, P Gaufillet, S Heim, F Vernadat (2010). Formal verification of AADL models with Fiacre and Tina. *ERTS 2010 proceedings*.

[10] Beugnard, A, J-M Jézéquel, N Plouzeau, D Watkins (1999). Making components contract aware. *IEEE Computer*, 32(11): 38-45.

[11] Broy, M, Gh Ştefănescu (1996). The algebra of stream-processing functions. Technical report (SFB-Bericht) 342/11/96 A, Technische Universität München.

[12] Broy, M, K Stølen (2001). Specification and development of interactive systems: FOCUS on streams, interfaces and refinement. Heidelberg: Springer.

[13] Burch, J R (1991). Verifying liveness properties by verifying safety properties (extended abstract). In: Clarke, Kurshan (eds), *Computer-aided verification* (CAV ’90). LNCS 531: 224-232. DOI 10.1007/BFb0023736.

[14] Burge, W H (1975). Stream processing functions. *IBM Journal of Research and Development*, 19(1). DOI 10.1147/rd.191.0012.

[15] Campos, J C, J Machado (2013). A specification patterns system for discrete event systems analysis. *International Journal of Advanced Robotic Systems*, 10. DOI 10.5772/56412.

[16] Cassez, Fr, K Larsen (2000). The impressive power of stopwatches. In: Palamidessi (ed), *CONCUR 2000*. LNCS 1877: 138-152.

[17] Chaki, S, E M Clarke, J Ouaknine, N Sharygina, N Sinha (2004). State/event-based software model checking. In: Boiten, Derrick, Smith (eds), *IFM 2004*. LNCS 2999, 128-147.

[18] Cimatti, A, St Tonetta (2012). A property-based proof system for contract-based design. *Proceedings of the 38th Euromicro Conference on Software Engineering and Advanced Applications*: 21-28. DOI 10.1109/SEAA.2012.68.

[19] Clarke, E, A Biere, R Raimi, Y Zhu (2001). Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19: 7-34.

[20] Cuenot, Ph, D Chen, S Gérard, H Lönn, M-O Reiser, D Servat, C-J Sjöstedt, R T Kolagari, M Törngren, M Weber (2007). Managing complexity of automotive electronics using the EAST-ADL. 12th International conference on engineering complex computer systems (ICECCS 2007).

[21] Damm, W, H Hungar, B Josko, Th Peikenkamp, I Stierand (2011). Using contract-based component specifications for virtual integration

testing and architecture design. *Design, Automation & Test in Europe* (DATE 2011). DOI 10.1109/DATE.2011.5763167.

[22] Dwyer, M B, G S Avrunin, J C Corbett (1999). Patterns in property specifications for finite-state verification. *ICSE '99 proceedings*: 411-420. DOI 10.1145/302405.302672.

[23] Frey, P Chr (2010). A timing model for real-time control-systems and its application on simulation and monitoring of AUTOSAR systems. Dissertation, Universität Ulm.

[24] Gafni, V (2008). Contract specification language (CSL). SPEEDS project deliverable D.2.5.4.

[25] Gafni, V. About the compilation of CSL, a real-time, pattern-based specification language. Slide set (unofficial). http://slidefinder.net/c/csl_tau_talk_july/15237881, 2014-04-02.

[26] Goknil, Ar, J Suryadevara, M-A Peraldi-Frati, F Mallet (2013). Analysis support for TADL2 timing constraints on EAST-ADL models. In: Drira (ed), *ECSA 2013*. LNCS 7957: 89-105.

[27] Harel, D (1987). Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8: 231-274.

[28] Harel, D, B Rumpe (2004). Meaningful modeling: what's the semantics of "semantics"?. *IEEE Computer*, 37(10): 64-72.

[29] Henzinger, Th, P W Kopke (1994). Undecidability results for hybrid systems. *Workshop on Hybrid Systems and Autonomous Control*, Ithaca.

[30] Henzinger, Th, Sh Qadeer, Sr K Rajamani (1998). You assume, we guarantee: methodology and case studies. *Computer Aided Verification proceedings*. LNCS 1427: 440-451.

[31] http://aadl.info, 2014-01-14.

[32] http://autofocus.in.tum.de/, AutoFocus 3 Wiki, 2013-12-16.

[33] http://btc-es.de/, 2014-04-01.

[34] http://cs.cmu.edu/~modelcheck/smv.html, 2014-04-02.

[35] http://esterel-technologies.com/products/scade-suite/, 2014-04-01.

[36] http://projects.laas.fr/tina/, "The TINA toolbox homepage", 2013-07-01.

[37] http://prover.com/products/prover_plugin/, 2013-08-15.

[38] http://speeds.eu.com/, 2013-02-28.

[39] http://spinroot.com, 2014-03-31.

[40] http://uppaal.org, 2014-04-02.

[41] http://vlsi.colorado.edu/~vis/, 2014-02-28.

[42] http://www.atesst.org/home/liblocal/docs/ATESST_EAST-ADL2Overview.pdf, 2013-03-10.

[43] http://www-cs-faculty.stanford.edu/~uno/faq.html, Donald E Knuth's homepage, Stanford University. 2014-03-07.

[44] https://www.cl.cam.ac.uk/research/hvg/Isabelle/, 2013-08-12.

[45] IEEE (2010). Standard for Property specification language (PSL). Reference no Std 1850-2010.

[46] ISO/IEC/IEEE (2011). International standard 42010: Systems and software engineering—Architecture description. Reference no 42010:2011(E).

[47] Johnsen, A, Kr Lundqvist (2011). Developing dependable software-intensive systems: AADL vs. EAST-ADL. In: Romanovsky, Vardanega (eds), *Ada-Europe 2011*. LNCS 6652: 103-117.

[48] Kaiser, B (2006). State/event fault trees. Dissertation, Technische Universität Kaiserslautern.

[49] Kaiser, B, C Gramlich, M Förster (2007). State/event fault trees—A safety analysis model for software-controlled systems. *Reliability Engineering & System Safety*, 92(11): 1521-1537.

[50] Kindler, E, L Petrucci (2009). Towards a standard for modular Petri nets: a formalisation. In: Franceschinis, Wolf (eds), *PETRI NETS 2009*. LNCS 5606: 43-62.

[51] Krenn, W, D Nickovic, L Tec (2013). Incremental language inclusion checking for networks of timed automata. In: Braberman, Fribourg (eds), *FORMATS 2013*. LNCS 8053: 152-167.

[52] Krogh, B H, St Kowalewski (1996). State feedback control of condition/event systems. *Mathematical and Computer Modelling*, 23(11/12): 161-173.

[53] Kupferman, O, R Lampert (2006). On the construction of finite automata for safety properties. In: Graf, Zhang (eds), *ATVA 2006*. LNCS 4218: 110-124.

[54] Marechal, A, D Buchs (2012). Modular extensions of Petri nets: a survey. Technical report 218, Centre Universitaire d'Informatique, Université de Genève.

[55] Miyazawa, A, A Cavalcanti (2014). Refinement-based verification of implementations of Stateflow charts. *Formal Aspects of Computing*, 26: 367-405. DOI 10.1007/s00165-013-0291-6.

[56] Mondragon, O A, A Q Gates (2004). Supporting elicitation and specification of software properties through patterns and composite propositions. *International Journal of Software Engineering and Knowledge Engineering*, 14(1): 21-41.

[57] Müller-Lerwe, A, R Busch, Th Rambow, U Christen, U Gussen, M Kees (2009). Safety aspects on a micro-hybrid vehicle with manual gearbox. *SAE International Journal of Passenger Cars—Electronic and Electrical Systems*, 1(1): 26-37.

[58] Mzid, R, Ch Mraidha, J-Ph Babau, M Abid (2012). Real-time design models to RTOS-specific models refinement verification. *ACES-MB '12 proceedings*.

[59] Novak, G S (1995). Conversion of units of measurement. *IEEE Transactions on Software Engineering*, 21(8): 651-661.

[60] Peraldi-Frati, M-A, A Goknil, J De Antoni, J Nordlander (2012). A timing model for specifying multi clock automotive systems: the timing augmented description language V2. *17th International Conference on Engineering of Complex Computer Systems*. DOI 10.1109/ICECCS.2012.5.

[61] Qureshi, T N, D-J Chen, M Törngren (2012). A timed automata-based method to analyze EAST-ADL timing constraint specifications. In: Vallecillo & al (eds), *ECMFA 2012*. LNCS 7349: 303-318.

[62] Ringert, J O, B Rumpe (2011). A little synopsis on streams, stream-processing functions and state-based stream processing. *International Journal of Software and Informatics*, 5(1-2): 29-53.

[63] Salamah, S, A Q Gates, Vl Kreinovich (2012). Validated templates for specification of complex LTL formulas. *Journal of Systems and Software*, 85(8): 1915-1929.

[64] Spichkova, M (2008). Refinement-based verification of interactive real-time systems. *Electronic Notes in Theoretical Computer Science*, 214: 131-157. DOI 10.1016/j.entcs.2008.06.007.

[65] Sreenivas, R S, B H Krogh (1991). On condition/event systems with discrete state realizations. *Discrete Event Dynamic Systems: Theory and Applications,* 1: 209-236. Boston: Kluwer.

[66] Stephenson, A G, L S LaPiana, D R Mulville, P J Rutledge, Fr H Bauer, D Folta, Gr A Dukeman, R Sackheim, R Norvig (1999). Mars Climate Orbiter mishap investigation board, phase I report. National Aeronautics and Space Administration (NASA). ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf, 2014-01-29.

[67] Szyperski, C (2002). *Component software: beyond object-oriented programming*, 2nd edition. Boston: Addison-Wesley.

[68] Yang, N, X Guo, W Wang (2012). Formal verification of a UML state chart diagram with Uppaal. *International Journal of Hybrid Information Technology*, 5(4): 55-60.

Appendix A—Structure of the Stop/start system

Figure 40 and Figure 41 give an overview of the part of the Stop/start system considered in this report. “Empty” ports indicate variables that have been left out because they played no role in verification conditions or their refinements.

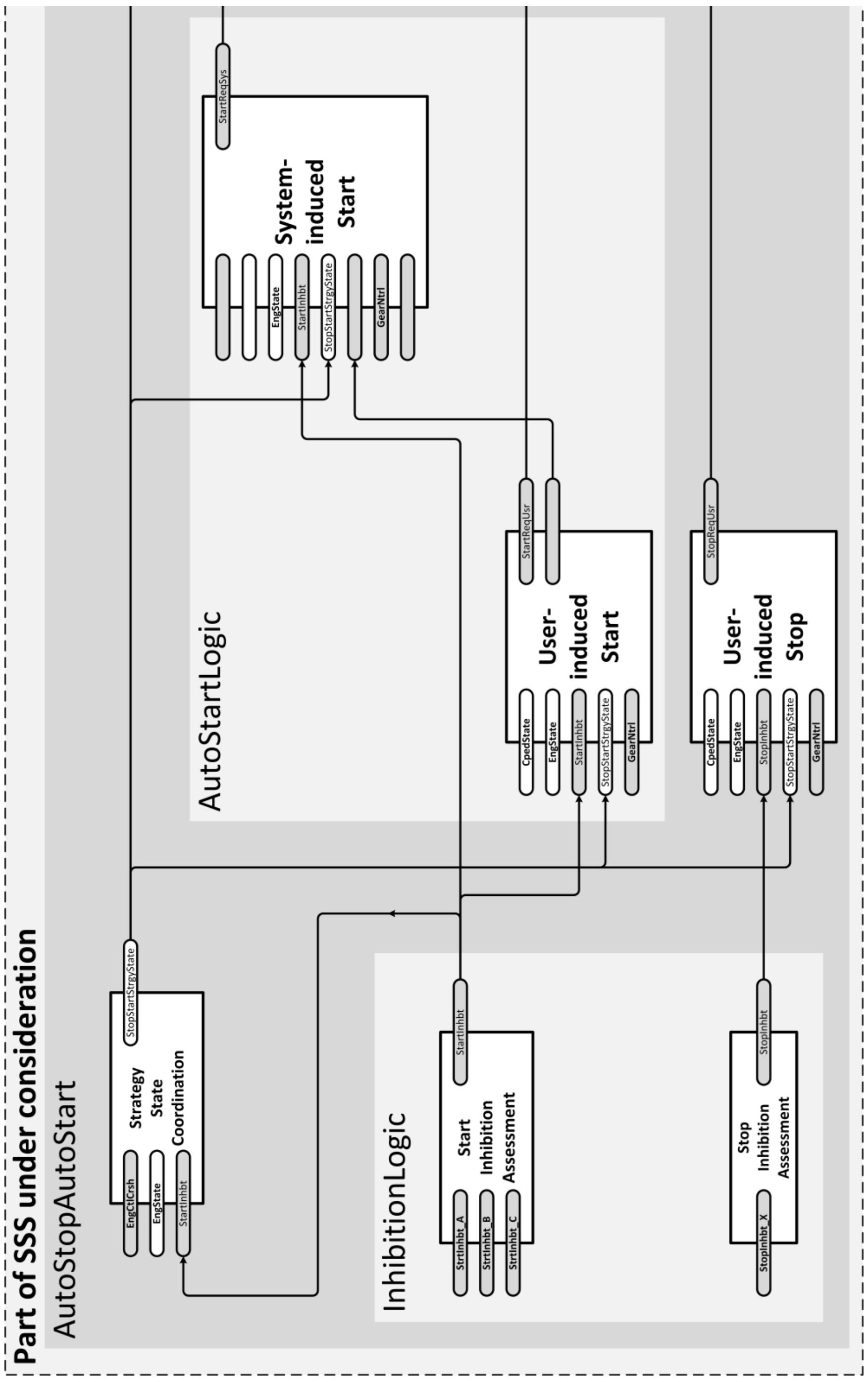


Figure 40. Overview of the Stop/start system (upstream part).

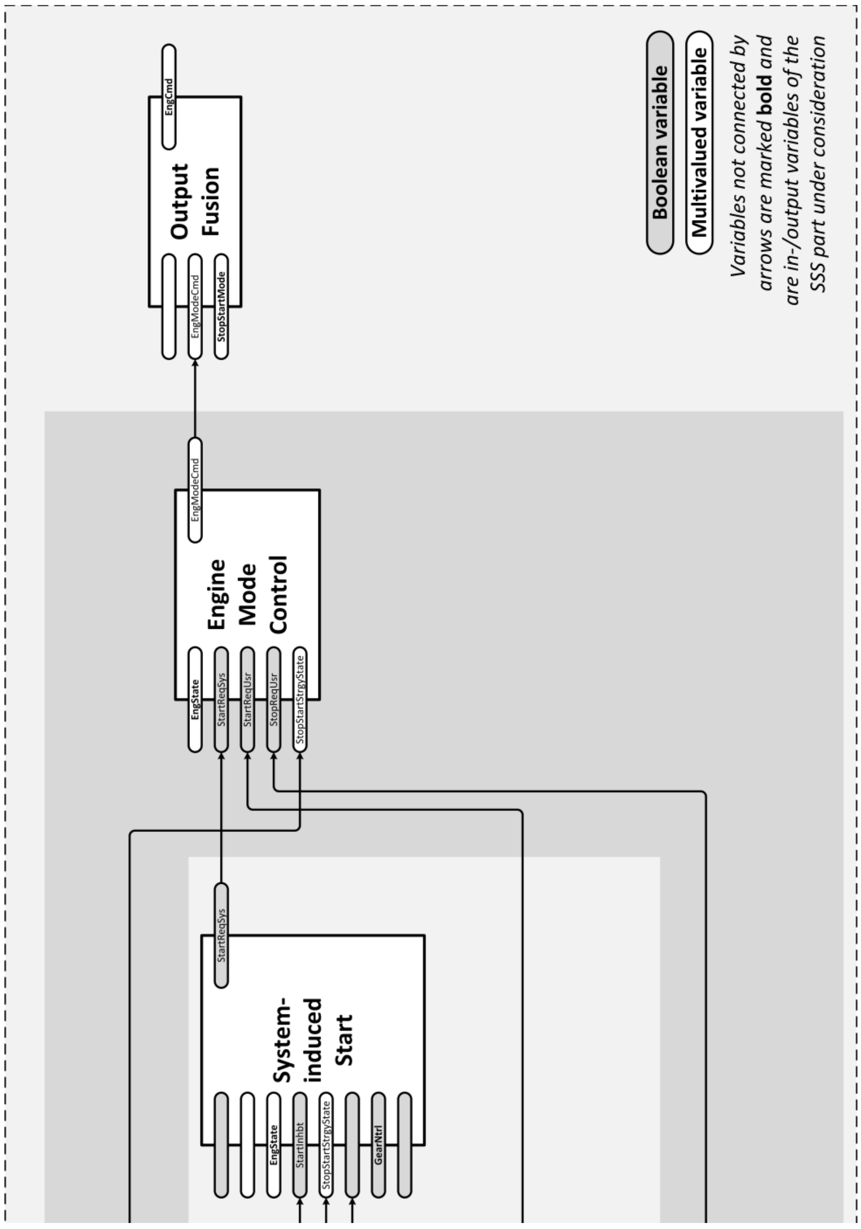


Figure 41. Overview of the Stop/start system (downstream part).

Appendix B—UPPAAL translation of verification condition 2

For the translation to UPPAAL of the STR_UT specification of verification condition 2, ten templates were used, depicted in the following. Note that, even though they are called TA, not all automata do indeed have a clock.

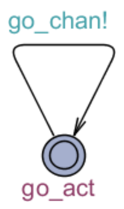


Figure 42. The "go_template" TA.

Instantiations of the *go_template* ensure that module translations immediately react to a change of input at their state inports.

Signature. `go_template(void).`

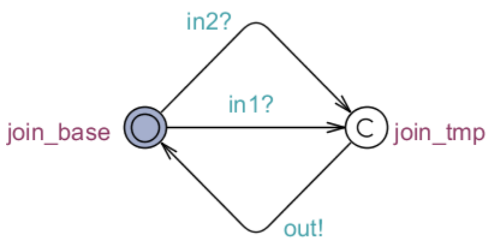


Figure 43. The "join" TA.

The *join* TA template represents the STR_UT *Union* module.

Signature. `join(broadcast chan &in1, broadcast chan &in2, broadcast chan &out).`

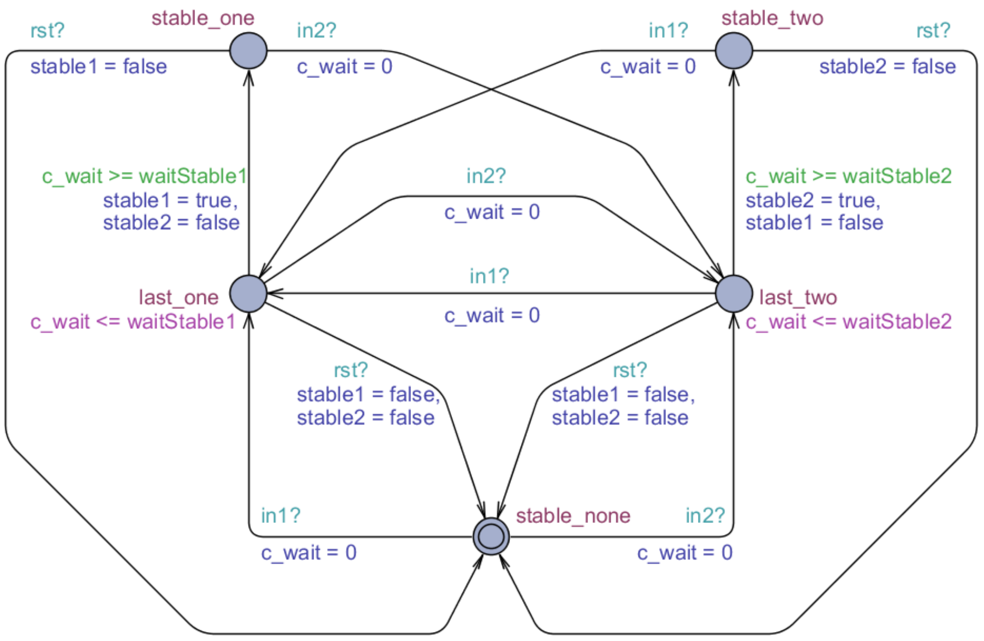


Figure 44. The "stable_debouncer" TA.

The *stable_debouncer* represents the STR_UT Debouncer module with event inports.

Signature. `stable_debouncer(int waitStable1, int waitStable2, broadcast chan &in1, broadcast chan &in2, broadcast chan &rst, bool &stable1, bool &stable2).`

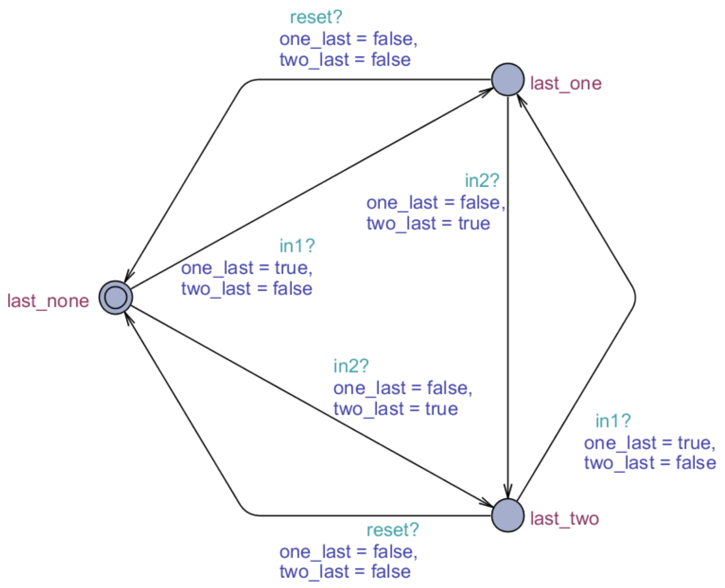


Figure 45. The "last" TA.

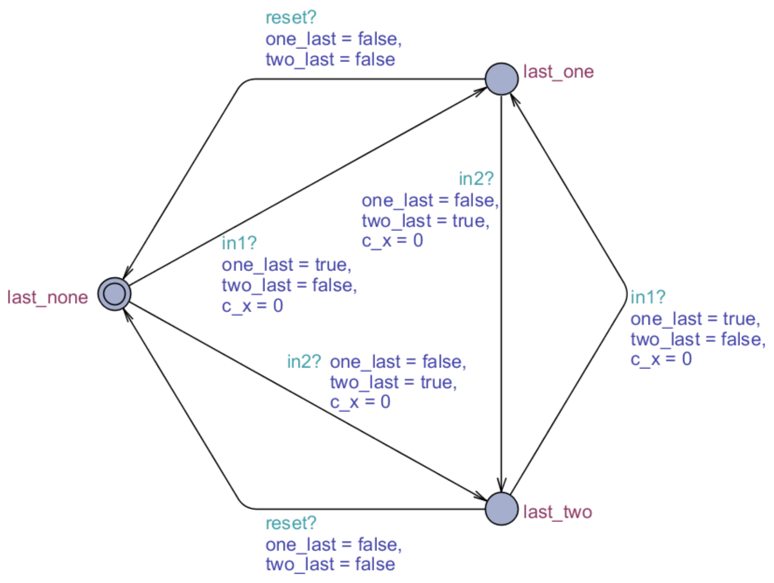


Figure 46. The "last_delay" TA.

last and *last_delay* represent the STR_UT *Last* module, with *last_delay* including, as a model simplification, a timer measuring the duration of a violation of the system guarantee.

Signature. `last[_delay](broadcast chan &in1, broadcast chan &in2, broadcast chan &reset, bool &one_last, bool &two_last).`

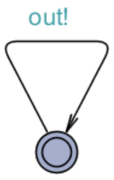


Figure 47. The "anytime_generator" TA.

The *anytime_generator* is the template for environmental input events.

Signature. `anytime_generator(broadcast chan &out).`

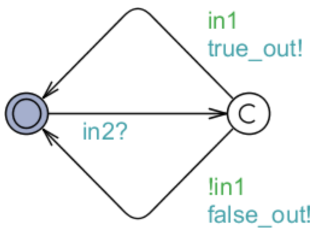


Figure 48. The "when" TA.

when represents the *When* primitive of STR_UT.

Signature. `when(bool &in1, broadcast chan &in2, broadcast chan &false_out, broadcast chan &true_out).`

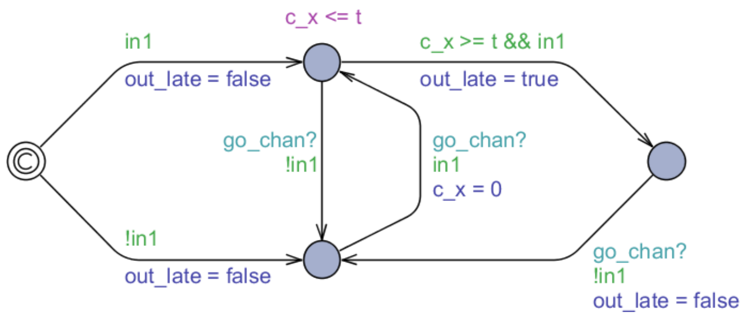


Figure 49. The "at_most" TA.

The *at_most* TA translates the STR_UT *Δt@Most* module.

Signature. `at_most(bool & in1, bool &out_late, int t).`

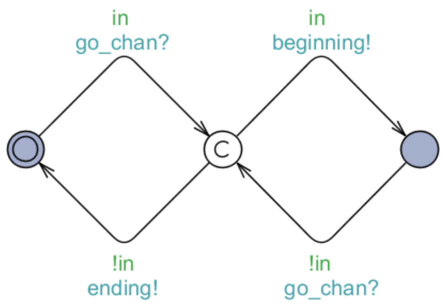


Figure 50. The "upon" TA.

upon represents the STR_UT *Upon* module.

Signature. `upon(bool &in, broadcast chan &beginning, broadcast chan &ending).`

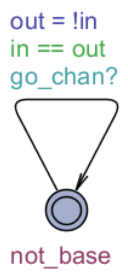


Figure 51. The "_not" TA.

_*not* performs negation and stands for the *Else* outport of the STR_UT *If* module. In the context of verification condition 2 it enables the reuse of the "When-Catch/Last" structure in the guarantee of Engine mode control (where in the STR_UT model of Figure 34 the *False* outport of *When* is connected instead of *True*).

Signature. `_not(bool &in, bool &out).`

System declaration and composition

```
// System description and template instantiation.
// Broadcast channels represent event ports.
broadcast chan Activate;
broadcast chan Deactivate;
broadcast chan Disinhibit;
broadcast chan Inhibit;
broadcast chan Join_out;
broadcast chan Last_in2;
broadcast chan Last_in21;
broadcast chan Last_in22;
broadcast chan Last_rst;
broadcast chan Last_rst1;
broadcast chan Last_rst2;
broadcast chan Last_rst3;
broadcast chan Last_rst4;
broadcast chan StableDebouncer_rst;
broadcast chan StableDebouncer_rst1;
broadcast chan StableDebouncer_rst2;
broadcast chan Start;
broadcast chan Upon_beginning;
broadcast chan Upon_beginning1;
broadcast chan Upon_beginning2;
broadcast chan Upon_beginning3;
broadcast chan Upon_ending;
broadcast chan Upon_ending1;
broadcast chan Upon_ending2;
broadcast chan Upon_ending3;
broadcast chan When_false;
broadcast chan When_false1;
broadcast chan When_false2;
broadcast chan When_false3;
broadcast chan When_true;
broadcast chan When_true1;
broadcast chan When_true2;
broadcast chan When_true3;

// Variables representing state ports.
bool Last_oneLast;
bool Last_oneLast1;
bool Eng_Violated;
bool Last_twoLast;
bool Last_twoLast1;
bool Last_twoLast2;
bool Last_twoLast3;
bool Last_twoLast4;
bool Not_out;
bool StableDebouncer_stable1;
bool StableDebouncer_stable11;
bool StableDebouncer_stable12;
bool StableDebouncer_stable2;
bool StableDebouncer_stable21;
bool StableDebouncer_stable22;
bool Stc_Violated;
bool Sys_Violated;
bool TAtMost_late;

// The module instantiations.
StableDebouncer3 =
   stable_debouncer(3, 3, Deactivate, Activate,
      StableDebouncer_rst2,
      StableDebouncer_stable12,
      StableDebouncer_stable22);

StableDebouncer1 =
   stable_debouncer(2, 10, Disinhibit, Inhibit,
      StableDebouncer_rst, StableDebouncer_stable1,
      StableDebouncer_stable2);

StableDebouncer2 =
   stable_debouncer(3, 3, Disinhibit, Inhibit,
      StableDebouncer_rst1,
      StableDebouncer_stable11,
      StableDebouncer_stable21);
```

```
Last2 =
   last(When_false1, Upon_beginning1, Last_rst2,
      Last_oneLast, Last_twoLast2);

ForbiddenStart_Last2 =
   last(When_true3, Last_in22, Last_rst4,
      Eng_Violated, Last_twoLast4);

Last1 =
   last(Join_out, Last_in21, Last_rst1,
      Stc_Violated, Last_twoLast1);

Last3 =
   last(Deactivate, Activate, Last_rst3,
      Last_oneLast1, Last_twoLast3);

ForbiddenStart_Last =
   last_delay(When_true, Last_in2, Last_rst,
      Sys_Violated, Last_twoLast);

When1 =
   when(Last_oneLast1, Upon_beginning,
      When_false1, When_true1);

ForbiddenStart_When2 =
   when(Not_out, Start, When_false3, When_true3);

ForbiddenStart_When =
   when(StableDebouncer_stable2, Start,
      When_false, When_true);

When2 =
   when(StableDebouncer_stable21, Upon_ending2,
      When_false2, When_true2);

Join1 =
   join(Upon_beginning3, When_true2, Join_out);

go_inst = go_template();

Upon1 =
   upon(StableDebouncer_stable21,
      Upon_beginning, Upon_ending);

Upon2 =
   upon(Last_oneLast1,
      Upon_beginning1, Upon_ending1);

Upon3 =
   upon(Last_oneLast1,
      Upon_beginning2, Upon_ending2);
```

```
Upon4 =
   upon(TAtMost_late,
      Upon_beginning3, Upon_ending3);

DeactivateGenerator =
   anytime_generator(Deactivate);

DisinhibitGenerator =
   anytime_generator(Disinhibit);

ActivateGenerator = anytime_generator(Activate);

InhibitGenerator = anytime_generator(Inhibit);

StartGenerator = anytime_generator(Start);

Not1 = _not(StableDebouncer_stable22, Not_out);

TAtMost1 = at_most(Last_oneLast, TAtMost_late, 3);

// The system composition. The go_inst TA
// is provided with the higest priority.
system
StableDebouncer3, StableDebouncer1,
StableDebouncer2, Last2, ForbiddenStart_Last2,
Last1, Last3, ForbiddenStart_Last, When1,
ForbiddenStart_When2, ForbiddenStart_When, When2,
Join1, Upon1, Upon2, Upon3, Upon4,
DeactivateGenerator, DisinhibitGenerator,
ActivateGenerator, InhibitGenerator,
StartGenerator, Not1, TAtMost1 < go_inst;
```

Queries & TA network

The sanity check in UPPAAL syntax was

```
E<> Sys_Violated && ForbiddenStart_Last.c_x > 0.
```

The actual verification query was

```
A[] !Sys_Violated || ForbiddenStart_Last.c_x == 0
|| Stc_Violated || Eng_Violated.
```

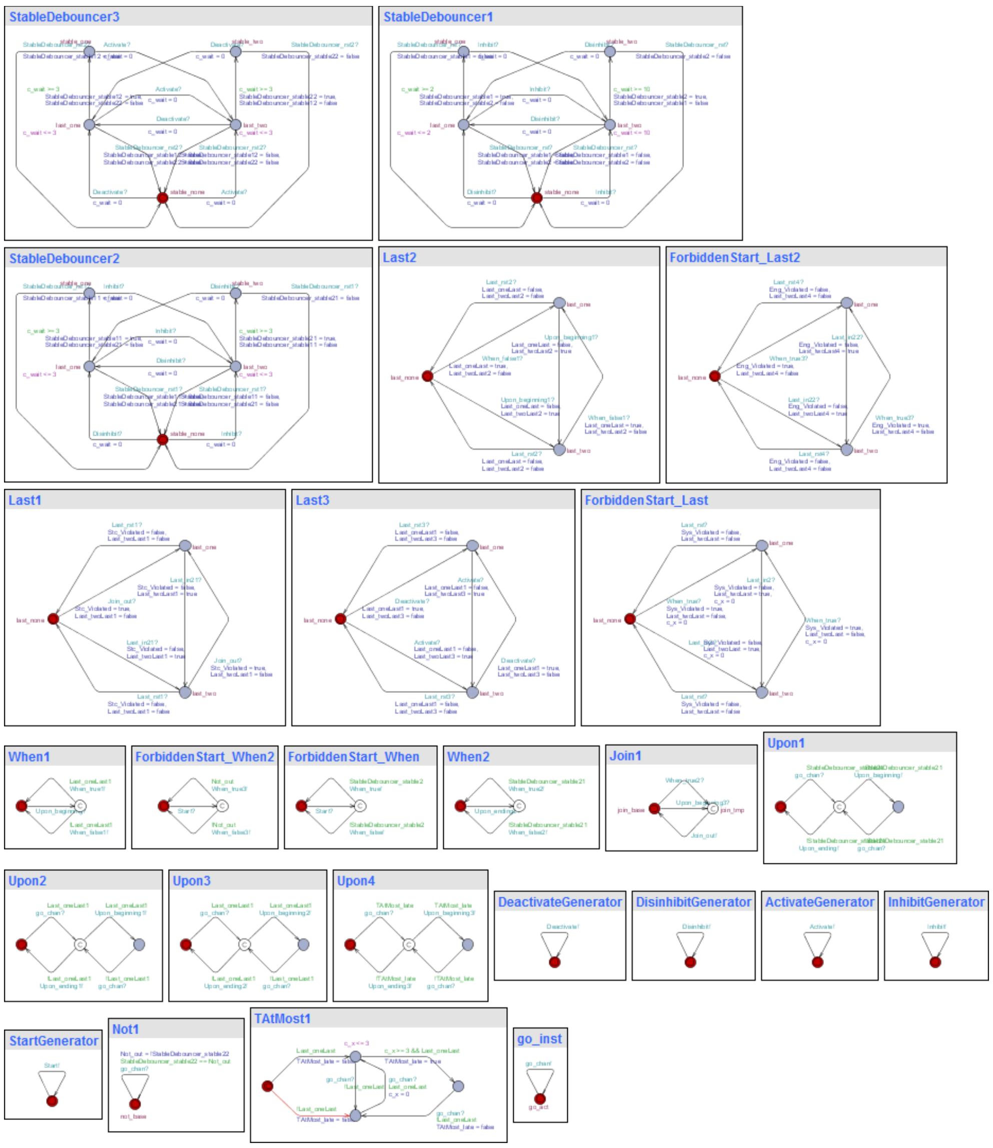


Figure 52. Screenshot of the TA network for verification condition 2 in UPPAAL.

Appendix C—TPN specifications of STR_UT primitives

We assume the concept of Time Petri nets (TPN) as implemented by the Tina (Time Petri net analyser) toolbox [36], featuring inhibitor and read arcs, timed transitions and transition priorities. For now we restrict timings to singular intervals. Structured automata augment the description of TPN by modularity. The legend in Table 4 describes TPN model elements.

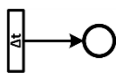
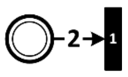
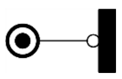
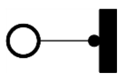
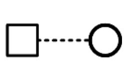
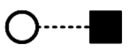
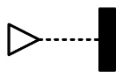
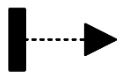
Timed transition, flow arc, place	Multi-token place, weighted flow arc, prioritised timeless transition	Marked place, inhibitor arc, timeless transition	Place, enabler arc, timeless transition
			
State inport, published place	Published place, state outport	Event inport, published transition	Published transition, event outport
			

Table 4. Elements of modular TPN.

Additionally, this section gives examples of how STR_UT atomic modules are defined in the TPN setting. During translation of an invariant to TINA format these PN fragments are merged by simple place fusion (“Fusion”) and transition fusion (“Trigger”) of those nodes that are published by their interface and, through it, connected to other modules.

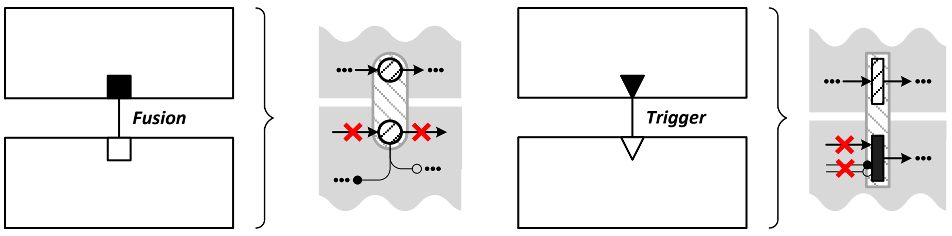


Figure 53. Merge rules for place and transition fusion of TPN representations of STR_UT modules.

Substructures of stateful nets effecting a reset to the initial state are highlighted with a grey background.

To preserve causality after merging, the PN representations of STR_UT modules obey so-called “merge rules”: places representing state inports must not be attached to flow arcs; transitions representing event inports can have no arc type attached to them except outflow arcs (Figure 53).

This presentation is exemplary and contains just a part of STR_UT modules. For orientation places are labelled with greek letters. Note that these nets are primarily meant to serve as a reference for module semantics, not for direct use in modelling.[8]

Or

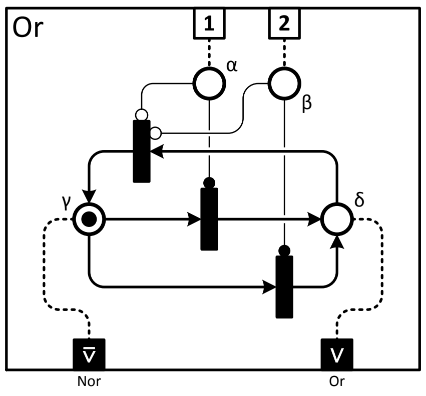


Figure 54. *Or* as PN.

Whenever both inputs are *false*, α and β are unmarked and place γ is marked (= the *Nor* outport is *true*). Otherwise, place γ is unmarked and place δ marked instead (= the *Or* outport is *true*).

[8] Concept and visualisation of the modular-PN representation of STR_UT modules owe to Condition/event systems [52] [65] as well as to the related State/event fault trees [48] [49].

If

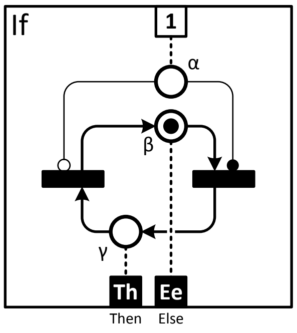


Figure 55. *If* as PN.

Whenever place α is unmarked (= the inport is *false*), the γ place is unmarked and the β place is marked (= the *Then* outport is *false* and the *Else* outport is *true*). Otherwise, the γ place is marked and the β place is unmarked (= the *Then* outport is *true* and the *Else* outport is *false*).

Union

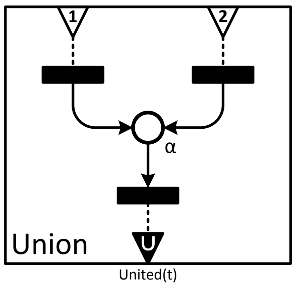


Figure 56. *Union* as PN.

Whenever one of the input transitions fires (= there is an incoming event) at a time *t*, place α gets marked for a time period of length zero, and the token is immediately consumed by the output transition, producing an outgoing event at time *t*.

When

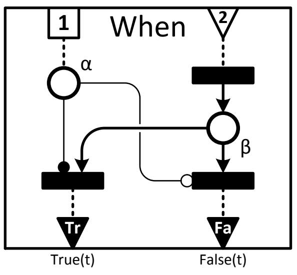


Figure 57. *When* as PN.

When place α is marked (= the state inport is *true*) and there is an incoming event the transition associated to the *True* outport fires, producing a *True* event. If place α is unmarked when an input event occurs, a *False* output event occurs.

Upon

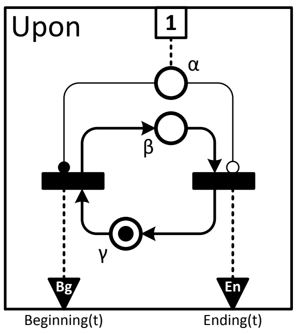


Figure 58. *Upon* as PN.

When place α becomes marked (= the inport condition becomes *true*) the transition associated with the *Beginning* outport fires, indicating a *Beginning* event. When place α becomes unmarked, an *Ending* event is produced at the corresponding outport.

Note that this PN is structurally identical to that of the *If* module. All four meanings “Then”/”Else”/”Beginning”/”Ending” could be realised by a single module.

Last

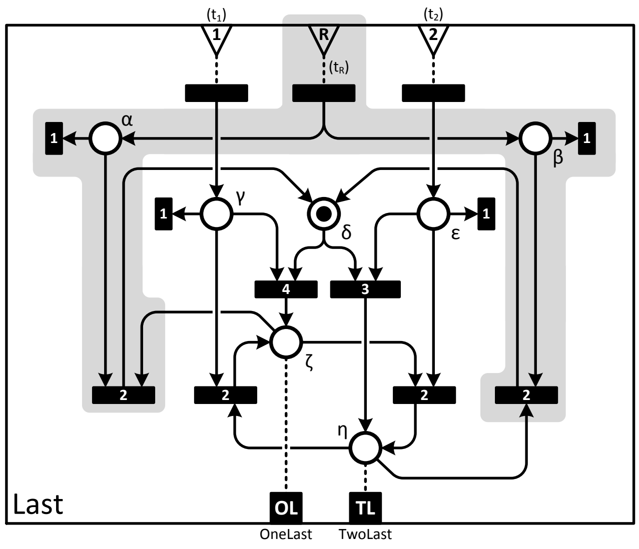


Figure 59. *Last* as PN.

In the initial state only place δ is marked. When an event at inport 1 happens, the associated transition fires, marking place γ. With places γ and δ marked, the transition with priority “4” fires, marking place ζ. Accordingly, the *OneLast* outport becomes true. In this state any further event at inport 1 marks place γ as well, but this token gets consumed immediately by the transition with priority “1” of γ, and outport values do not change. In a similar way, an event at inport 1 marks place η, making the *TwoLast* outport true. A reset event marks place δ again and removes any other tokens.

nFloor

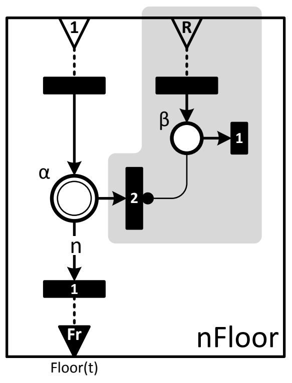


Figure 60. *nFloor* as PN.

Whenever the regular inport event happens, a token is added to the α place. As soon as the number of tokens in α reaches the arc weight *n*, the transition associated with the outport fires, producing a *Floor* event. Any time a reset event happens, place α is cleared of tokens, and the module returns to the initial state.

Δt@Most

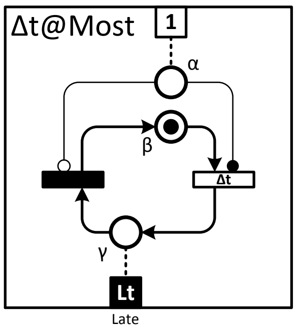


Figure 61. *Δt@Most* as TPN.

When the inport becomes *true* (= place α gets marked), the timed transition becomes enabled. After the α place has been continuously marked for a time *Δt* the timed transition fires, removing the token from place β and marking place γ (= the *Late* outport becomes *true*). As soon as α becomes unmarked again the module returns to its initial state. In case the marking of α is removed before *Δt* expires, the timed transition is disabled, returning the module to the initial state as well.

Note that this module is structurally similar to the *If* and *Upon* modules, but since it has a timed transition, which neither of the other modules does, it cannot be realised by the same PN.

ΔtDelay

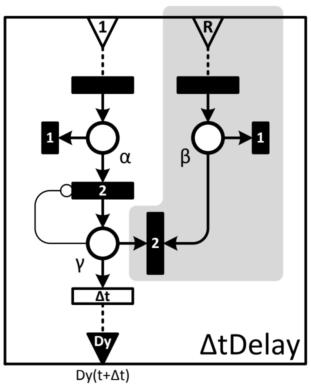


Figure 62. *ΔtDelay* as TPN.

When an event happens at the regular inport and place γ is unmarked, γ becomes marked, enabling the timed transition and preventing further firings of its input transition. When the time *Δt* expires and no reset event has happened in the meantime the timed transition fires, producing an output event *Dy*. If γ is marked, any reset event removes the token and disables the timed transition. As long as γ remains marked, additional regular input events will be ignored, new tokens in α being consumed by the transition with priority “1”.

Never

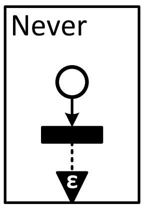


Figure 63. The event inport terminator.

While state ports of and inside a STR_UT module can remain unconnected, and also event outports do not need a connection, unconnected event inports have to be terminated. This is done by connecting them to a *Never* module containing just an empty place, thus effectively disabling the transition.

Bisher in der FAT-Schriftenreihe erschienen (ab 2010)

Impressum

Herausgeber	FAT Forschungsvereinigung Automobiltechnik e.V. Behrenstraße 35 10117 Berlin Telefon +49 30 897842-0 Fax +49 30 897842-600 www.vda-fat.de
ISSN	2192-7863
Copyright	Forschungsvereinigung Automobiltechnik e.V. (FAT) 2015

VDA Verband der Automobilindustrie

FAT Forschungsvereinigung Automobiltechnik

Behrenstraße 35
10117 Berlin
www.vda.de
www.vda-fat.de