Self-Adaptive Automata*

Aimee Borda
Trinity College Dublin
Dublin, Ireland
bordaa@tcd.ie

Vasileios Koutavas
Trinity College Dublin
Dublin, Ireland
Vasileios.Koutavas@scss.tcd.ie

ABSTRACT

Self-adaptive systems aim to efficiently respond to a wide range of changes in their operational environment by dynamically altering their behaviour. Such systems are typically comprised of a base system, implementing core functionality, and an adaptation decision process, which determines how the base system must change at different points in its execution. These two components coordinate through a set of *adaptation events*: a set of execution points of the former where the latter is invoked. The pattern of these events is crucial for the overall system to achieve (a) correctness against specific requirements, and (b) efficiency of system resources. Existing techniques for modelling self-adaptive systems usually hardcode adaptation events within the base system or the adaptation decision process. This limits system designers in discovering correct and optimal patterns of adaptation events, as changing those involves significant changes in the model. In this work we present Self-Adaptive Automata, an abstract modelling framework which decouples adaptation event patterns from the descriptions of base systems and adaptation decision processes. In our framework, base systems expose execution points where adaptation *may happen*—in the most general case this can include all system states—and adaptation decision processes are parametric to these points. A distinct automaton then pinpoints when in the system adaptation *must happen*. Using this framework system designers can experiment with different adaptation event patterns, without modifications to the base system or the adaptation decision process, and discover correct and efficient patterns. We show that our framework is compatible with traditional verification tools by providing an adequate translation from Self-Adaptive Automata into FDR, in which correctness against requirements can be verified. We also prove that, although our automata framework includes dynamic self-modifying features, it corresponds to standard models of computation. We illustrate the use of our framework through a use case of a self-adaptive system of autonomous search-and-rescue rovers.

*This work was supported by Science Foundation Ireland grant 13/RC/2094 (Lero).

FormaliSE '18, June 2, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5718-0/18/06…$15.00
https://doi.org/10.1145/3193992.3194001

CCS CONCEPTS

• **Software and its engineering → System modeling languages**; **Formal software verification**; • **Theory of computation →** *Automata extensions*;

ACM Reference Format:
Aimee Borda and Vasileios Koutavas. 2018. Self-Adaptive Automata. In *FormaliSE '18: 6th Conference on Formal Methods in Software Engineering, June 2, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3193992.3194001

1 INTRODUCTION

There is an increasing number of systems engineered to be resilient, efficiently responding to a wide range of changes in their operational environments without human intervention [28, 29]. Self-adaptive systems aim to fulfil this goal by autonomously modifying their behaviour, reacting to environmental changes. Such systems consist of a base system and an adaptation decision process which modifies base system functionality. Designers of these systems must decide at which system execution points should adaptation be invoked in order for the system to operate efficiently and satisfy its requirements. These points can be as simple as timing events (e.g., every X seconds), can be triggered by specific system events, or even have a complex logic taking account the execution history and state of the system.

Existing modelling approaches for self-adaptive systems often distinguish the adaptation decision process from the base system, but encode adaptation event points directly in one, or both, of these components (e.g., [4, 13, 19, 25, 26, 36, 40]). The drawback of this approach is that the pattern of adaptation events becomes a cross-cutting concern which is scattered within the model of the system. Modifying this pattern in search for a correct and optimal model often requires significant changes to the description of the base system and/or the adaptation decision process, which has to be done in an ad-hoc manner.

However, modularising adaptation event patterns is not easy [15]. Although some systems may be able to support such events at every state, others must disallow them during critical sections of their execution. Moreover, adaptation decision processes may be able to operate only at a subset of the system states. For example, a collision avoidance system of autonomous vehicles, realised as an adaptation decision process, may assume that it is invoked at states where vehicles are not too close to each other, where reasonable direction changes can deter collisions. These aspects must be taken into account when adaptation event patterns are modelled or implemented, and the maximum degree of freedom for choosing these patterns should be allowed in each use case.

In this paper we propose an abstract model for self-adaptive systems, which we call Self-Adaptive Automata (SAA). With this automata-based model we abstract away as many implementation

details as possible of self-adaptive systems and focus only on the composition of their three main components: base system, adaptation decision process, and adaptation event pattern. This allows us to explore the design space of these components, keeping the others constant. In particular we can use this model to identify efficient adaptation event patterns in self-adaptive systems. Our model is also useful in proving system correctness (and thus the correctness of adaptation patterns) against requirements. We provide an adequate translation from SAA to FDR [16], where we can verify system properties. This high-level model can be used as an abstract description of self-adaptive systems, where key system aspects such as adaptation patterns can be explored, and key requirements verified. From that, a correct and efficient implementation could be derived through refinement, leveraging FDR's refinement infrastructure.

In SAA, adaptation events are exposed in execution traces by a novel, special-purpose ⋆-transition. The effect of a ⋆-transition is the modification of the automaton's entire transition function (a modification of its behaviour), according to the adaptation decision process, encoded as a partial function within the model. This can capture *may-* and *must-adapt* events. We use may-adapt events in the modelling of base systems, exposing the states where the system can invoke the adaptation decision process. These events are optional because other transitions from the same states allow the system to skip adaptation. Must-adapt events are used in the definition of adaptation event patterns, which are distinct automata—potentially implementing complex logic—with states whose only outgoing transition is an adaptation (⋆-transition). Composition of a system model with an adaptation pattern model synchronises adaptation events, in effect enforcing the adaptation pattern to the system. The benefit of this approach is that we can change the adaptation pattern without changing the model of the whole system.

We illustrate the use of SAA through the use case of a self-adaptive system of autonomous search-and-rescue vehicles, inspired by related work [1, 11, 25, 30]. Using our model, we are able to encode the base system and the adaptation decision process independently, separating them from adaptation event patterns. We are thus able to explore radically different adaptation patterns, and prove their correctness through our translation to FDR.

Our model extends standard automata with a self-modifying feature. Related extensions have been shown to significantly enhance the base model of computation [33]. Here we prove that this is not the case with SAA, by showing a correspondence between SAA and the standard model of execution monitors [35]. This also shows that adaptation in our framework can enforce all *safety properties*.

The rest of the paper is organised as follows. The next section presents the use case of a self-adaptive system of autonomous vehicles. Section 3 gives a high-level overview of our modelling framework. Section 4 provides the technical details of SAA and discusses how our use case can be encoded in it. Section 5 defines a translation from SAA to CSP [21], which is FDR's core language, and shows that this translation is adequate. The same section details how the autonomous vehicles model can be verified in FDR through the translation. Section 6 describes the correspondence between the computation model of SAA and execution monitors. Finally, in section 7 we discuss related work and in section 8 our conclusions.

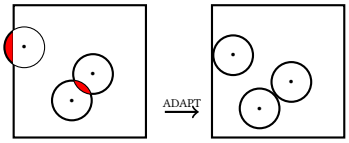

Figure 1: Self-Adaptive Autonomous Vehicles System.

2 SELF-ADAPTIVE AUTONOMOUS VEHICLES

Here we consider a simple self-adaptive system for search-and-rescue operations by unmanned vehicles, inspired by previous work [1, 11, 25, 30]. The system consists of a number of vehicles moving autonomously in a search area, and a central coordinator responsible for avoiding collisions between vehicles and vehicles escaping the search area. This is achieved by the vehicles reporting their position to the coordinator at specific points in their execution, and the coordinator running a centralised adaptation decision process, changing the behaviour of vehicles through remote commands when a collision is imminent. If two vehicles are closer together than a minimum critical distance the coordinator adjusts their behaviour so that they move further apart before continuing their normal movement. The following basic requirements must be satisfied by the system:

Req. 1: Vehicles must not collide with each other.
Req. 2: All vehicles must remain within the search area.
Req. 3: Every position in the search area must be eventually explored by the vehicles.

The base system here involves the actual vehicle movement, which can be random, or towards specific coordinates when instructed by the adaptation. The adaptation decision process can take as input the current coordinates of the vehicles, their speed, and direction, and issue adaptation commands to the vehicles, nudging them apart, when a possible collision is detected. Both of these components do not depend on the exact points where adaptation is invoked. It would thus be desirable to model our base system in a liberal way, allowing adaptation at virtually every state.[1]

Such a model would allow us to explore different adaptation event patterns that ensure system correctness, and evaluate them in terms of efficiency. Here we consider two such patterns.

Time-Triggered Implementation: A possible adaptation event pattern could invoke the adaptation procedure at predefined time intervals. This means that time must be explicit in the model which, in an automata-based model, can be done with *tock* self-loops in every state of the base system.

The adaptation decision process, assuming a minimum frequency of adaptation events, can compute new positions for vehicles that are in danger of collision or escaping the search area (see fig. 1) It should then be possible to verify that if adaptation events occur with at least that frequency, then Req. 1 and Req. 2 are satisfied. The third requirement, Req. 3, should also be satisfied, if vehicle movement is indeed random and the adaptation procedure does not always send vehicles to the same coordinates.

[1] If the adaptation decision manager makes assumptions about the system states it is invoked (e.g., a minimum distance between vehicles) then this should be taken into account by the allowed adaptation points.

Event-Triggered Implementation: An alternative implementation approach of the vehicles' coordinator is to invoke an adaptation procedure when vehicles are closer than a minimum distance and a collision is possible. With this adaptation pattern, the position of the vehicles must be monitored and once the minimum distance is reached for a pair of vehicles, an adaptation event occurs.

In large search areas this event-triggered adaptation pattern can be more efficient, in that it can preserve the system requirements with fewer invocations of the adaptation decision process.

3 OVERVIEW OF THE MODELLING FRAMEWORK

Two main components of a self-adaptive system are the base system and the adaptation decision process. As we discussed in the introduction, a third, equally important component is the adaptation event pattern. In our modelling framework, we use the abstract formalism of automata to express these components.

The base system is essentially a standard automaton, with states and a labelled-transition function. This expresses the regular behaviour of the system. To encode the adaptation decision process, we extend such automata with a special-purpose transition, annotated with a $\star$. During this transition, an adaptation function Π, which is part of the definition of our automata, inputs the current system state and outputs *a new transition function* and a new state for the system. The new transition function replaces the existing transition function of the base system, changing the automaton's behaviour and encoding adaptation. The new state enables the encoding of information sent from the adaptation decision process to the base system. Note that Π is partial, meaning that $\star$-transitions may not be defined for some system states. This allows us to encode *may-adapt* transitions; i.e., the possibility of adaptation at these states.

When a system designer models a self-adaptive system in SAA, the base behaviour of the system is first identified and encoded as a standard automaton. Then, inspecting the requirements of the system, a decision procedure is encoded as a function from states to transition functions and states. The states for which this function is defined get a $\star$-transition (can-adapt).

The last ingredient of the model is the adaptation pattern: when *must* the adaptation function be invoked during the execution of the automaton. In SAA this is encoded as a separate automaton which we call *adaptation automaton*, satisfying specific properties (see definition 4.3). After composition with the base system, the adaptation automaton determines the states where adaptation must happen, essentially pruning the outgoing transitions of can-adapt states of the base system. The encoding of adaptation event patterns as adaptation automata allows the system designer to encode multiple—simple and complex—adaptation event patterns, and evaluate them in terms of efficiency. Once a suitable adaptation automaton is established, the system can be verified for correctness through a translation to FDR.

4 SELF-ADAPTIVE AUTOMATA

We now present Self-Adaptive Automata (SAA), a formalism for modelling self-adaptive systems, and use them to model the autonomous vehicles example. Key aspects of SAA is the encoding of adaptation by dynamic modification of the transition function, and the inclusion of adaptation events in execution traces through a special-purpose $\star$ action. The latter is important for adaptation actions to synchronise during SAA composition. The formal definition of SAA is the following.

Definition 4.1 (Self-Adaptive Automata). A Self-Adaptive Automaton (SAA) $\mathcal{M}$ is a tuple $\langle Q, \Sigma, \Delta, q_0, \delta_0, \Pi\rangle$ where:

- Q is a set of states;
- Σ is a set of symbols; we let a range over Σ;
- $\Delta \in \mathcal{P}(Q \times \Sigma \rightharpoonup Q)$ is a set of transition functions, partially mapping states and symbols to states;
- $q_0 \in Q$ is the initial state;
- $\delta_0 \in \Delta$ is the initial transition function;
- $\Pi \in Q \rightharpoonup Q \times \Delta$ is the adaptation function, partially mapping states to states and transition functions.

In SAA, behaviour modification (i.e., adaptation) is an atomic action. It occurs as a single $\star$-event. This is a useful simplification when considering high-level models of SA systems, although it may not capture the behaviour of systems where adaptation is propagated non-atomically, perhaps for performance reasons. Moreover, the definition considers deterministic SAA in order to compare with EMs (cf. section 6) which are also deterministic. The translation to CSP (cf. section 5) can be adapted to non-deterministic SAAs.

The operational semantics of an SAA is defined as a labelled transition system (LTS) over configurations containing the current state and active transition function of the automaton. We use the symbol $\star$ to label adaptation transitions; we let $\Sigma_\star = \Sigma \cup \{\star\}$ and $a_\star$ range over $\Sigma_\star$.

$$\langle q, \delta\rangle \xrightarrow{a} \langle q', \delta\rangle \qquad \text{if } \delta(q, a) = q'$$

$$\langle q, \delta\rangle \xrightarrow{\star} \langle q', \delta'\rangle \qquad \text{if } \Pi(q) = \langle q', \delta'\rangle$$

This semantics shows that SAA configurations transition according to the function δ in the configuration, which can change by $\star$-transitions, modelling adaptation. These adaptation transitions make SAA an expressive framework for adaptive systems, while still remaining a standard computation model (section 6). Note traditional deterministic automata are a special case of SAA, with no $\star$-transitions.

Example 4.2. We now turn our attention to the example in section 2. We can model this example with an SAA

$$\mathcal{M}_1 = \langle Q, \Sigma, \Delta, q_0, \delta_0, \Pi\rangle$$

composed of the following elements:

$$\Sigma = \{goto.p_1.p_2 \mid p_1, p_2 \in Loc\}$$
$$Q = \{\langle p_1, p_2\rangle \mid p_1, p_2 \in Loc\}$$

Here we assume a set of locations *Loc* representing the possible positions of a vehicle, and a *distance* function $d(p_1, p_2)$ which gives the cartesian distance of any two points. We assume that there are positions every one unit on the horizontal and vertical axes. For simplicity we restrict our attention to a system with two vehicles, and consequently the transitions $goto.p_1.p_2$ track the positions of these vehicles over time. We use one state for each pair of positions of the two vehicles.

The initial transition function δ_0 allows the vehicles to move to any position which is at most one distance unit away from the current position; in effect vehicles can move one position to the left, right, up or down, or stay at the same position.

$$\delta_0(\langle p_1, p_2\rangle, goto.m_1.m_2) = \begin{cases} \langle m_1, m_2\rangle & \text{if } d(p_1, m_1) \le 1 \\ & \text{and } d(p_2, m_2) \le 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Recall from section 2 that if the coordinator deems that the vehicles are too close, it will issue a command for the vehicles to move further apart. This is captured here with a transition function where the two vehicles can only move to certain positions, before making any more moves. This transition function depends on the current positions of the vehicles, p_1 and p_2, and the positions that they must move to, l_1 and l_2, respectively. Thus we have a family of transition functions which depend on these parameters:

$$\delta_{\langle s_1, s_2\rangle \to \langle l_1, l_2\rangle}(\langle p_1, p_2\rangle, goto.m_1.m_2) =$$
$$\begin{cases} \langle m_1, m_2\rangle & \text{if } m_1 = l_1, m_2 = l_2, s_1 = p_1 \text{ and } s_2 = p_2 \\ \langle m_1, m_2\rangle & s_1 \neq p_1, s_2 \neq p_2 \text{ and } d(p_1, m_1) \le 1, d(p_2, m_2) \le 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

This function allows only a single move from $\langle s_1, s_2\rangle$; that to $\langle l_1, l_2\rangle$. From any other $\langle p_1, p_2\rangle$ all valid moves are allowed.

The model has an adaptation transition from every state.

$$\Pi(\langle p_1, p_2\rangle) =$$
$$\begin{cases} (\langle p_1, p_2\rangle,\ \delta_{\langle p_1, p_2\rangle \to \langle p_1', p_2'\rangle}) & \text{if } danger(\langle p_1, p_2\rangle) \\ & \text{and } safe(\langle p_1, p_2\rangle) = \langle p_1', p_2'\rangle \\ (\langle p_1, p_2\rangle,\ \delta_0) & \text{otherwise} \end{cases}$$

Adaptation relies on the auxiliary predicate *danger* which identifies the states where vehicles are too close to each other, and the function *safe* which, when given a dangerous position, returns safe positions that the vehicles can move to, as depicted in fig. 1.

Although adaptation is possible at every state, the system does not need to adapt after every move, provided that there is enough distance between vehicles, and between vehicles and the border. For example if $danger(\langle p_1, p_2\rangle)$ is true when p_1 and p_2 are at distance less than six units between them and less than three units from the border, and assuming *safe* moves vehicles to positions where the *danger* predicate is false, then the system can safely adapt every *two* transitions, guaranteeing no collisions.

As we will see in the next subsection, different adaptation automata can be defined independently from the base system and adaptation decision process, and be combined with them through automata composition.

4.1 Adaptation Automata

Here we model adaptation event patterns as a special class of SAA, which we call adaptation automata. We also define composition of SAA, used for enforcing an adaptation event patterns on system models.

An adaptation automaton pinpoints which adaptation moves *must* be performed. Consequently, it can be modelled by an SAA whose adaptation transitions are mandatory. This means that states with outgoing adaptation transitions have no other outgoing transition.

Moreover, we assume that a single adaptation move is powerful enough to give the system the desired behaviour. Therefore, adaptation policies, by definition, have no two consecutive $\star$-moves. This excludes the possibility of infinite adaptation moves and makes the number of possible consecutive adaptation moves deterministic (it is always one).

Definition 4.3 (adaptation automaton). We say an SAA

$$\mathcal{M} = \langle Q, \Sigma, \Delta, q_0, \delta_0, \Pi\rangle$$

is an adaptation automaton when for all $q \in Q$ and $\delta \in \Delta$ the following conditions hold:

(1) if $\langle q, \delta\rangle \xrightarrow{\star}\!\!\!\!\!\!/\ \ $ then $\langle q, \delta\rangle \xrightarrow{a}$, for all $a \in \Sigma$ and;
(2) if $\langle q, \delta\rangle \xrightarrow{\star} \langle q', \delta'\rangle$ then $\langle q, \delta\rangle \xrightarrow{a}\!\!\!\!\!\!/\ \ $, for all $a \in \Sigma$;
(3) if $\langle q, \delta\rangle \xrightarrow{\star} \langle q', \delta'\rangle$ then $\langle q', \delta'\rangle \xrightarrow{\star}\!\!\!\!\!\!/\ \ $

We will call a state with an outgoing $\star$-transition, an *adaptation state*, and any other state a *regular state*. The aim of the adaptation automaton is to pinpoint the adaptation points during a system's execution. The adaptation automaton should not influence the execution in any other way or include alternative execution paths which allow a system to bypass an adaptation. These two properties are respectively enforced by Conditions 1 and 2 in the above definition. Condition 1 ensures that if an adaptation is not to occur from the current state then the adaptation automaton must enable all Σ-transitions so as not to influence the system's execution. Condition 2, together with the fact that SAA are deterministic by definition, ensures that when adaptation automaton state that an adaptation is to happen, both the adaptation automaton and the base system have no alternative transition except the adaptation.

One of the benefits of SAAs is that they can be composed by automata intersection. In particular we will use this to compose models of systems with adaptation automata.

Definition 4.4 (SAA Composition). Let

$$\mathcal{M}_1 = \langle Q_1, \Sigma, \Delta_1, q_0', \delta_0', \Pi_1\rangle$$

and

$$\mathcal{M}_2 = \langle Q_2, \Sigma, \Delta_2, q_0'', \delta_0'', \Pi_2\rangle$$

be SAA. We define $\mathcal{M}_1 \cap \mathcal{M}_2$ to be the SAA:

$$\langle Q_1 \times Q_2, \Sigma, \Delta, (q_0', q_0''), \delta_0' \cap \delta_0'', \Pi\rangle$$

where $Q_1 \times Q_2$ is the cartesian product of the states and

$$\Delta = \{\delta_1 \cap \delta_2 | \delta_1 \in \Delta_1 \text{ and } \delta_2 \in \Delta_2\}$$
$$(\delta_1 \cap \delta_2)(\langle q_1, q_2\rangle, a) = \langle q_1', q_2'\rangle \quad \text{if } \delta_1(q_1, a) = q_1',\ \delta_2(q_2, a) = q_2'$$
$$\Pi(\langle q_1, q_2\rangle) = \langle (q_1', q_2'), \delta_1' \cap \delta_2'\rangle \quad \text{if } \Pi_1(q_1) = \langle q_1', \delta_1'\rangle,\ \Pi_2(q_2) = \langle q_2', \delta_2'\rangle$$

Theorem 4.5. *For any two SAAs $\mathcal{M}_1$ and $\mathcal{M}_2$:*

(1) if M_1 or M_2 is an adaptation automaton then $M_1 \cap M_2$ is an adaptation automaton;
(2) if M_1 and M_2 are finite then $M_1 \cap M_2$ is finite;
(3) SAA intersection is a commutative, associative and idempotent operation.

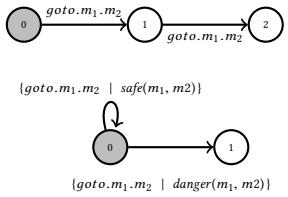

Figure 2: The transition function for $\mathcal{A}_1$ (top) and $\mathcal{A}_2$ (bottom)

Returning to the example of section 2 we note that a self-adaptive system may satisfy a requirement only under certain adaptation event patterns. For example, the vehicle system modelled in example 4.2 avoids collisions only if adaptation runs at least once every two system transitions.

Example 4.6. In section 2 we discussed two adaptation event patterns for autonomous vehicles. Here we encode them as adaptation automata as shown in fig. 2. The time-triggered adaptation automaton requires adaptation every two transitions. Let

$$\mathcal{A}_1 = \langle Q, \Sigma, \Delta, q_0, \delta_0, \Pi \rangle$$

where

$$Q = \bigcup_{i \in 0 \ldots 2} q_i \qquad \Sigma = \{goto.p_1.p_2 \mid p_1, p_2 \in Loc\} \qquad \Delta = \{\delta_0\}$$

$$\delta_0(q_i, goto.p_1.p_2) = \begin{cases} q_{i+1} & \text{if } i \in \{0, 1\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\Pi(q) = \begin{cases} \langle q_0, \delta_0 \rangle & \text{if } q = q_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Dually, the event-triggered adaptation automaton discussed in section 2 requires an adaptation when the vehicles becoming dangerously close to one another. Let

$$\mathcal{A}_2 = \langle Q, \Sigma, \Delta, q_0, \delta_0, \Pi \rangle$$

where

$$Q = \{q_0, q_1\} \qquad \Sigma = \{goto.p_1.p_2 \mid p_1, p_2 \in Loc\} \qquad \Delta = \{\delta_0\}$$

$$\delta_0(q, goto.p_1.p_2) = \begin{cases} q_0 & \text{if } q = q_0 \text{ and } safe(\langle p_1, p_2 \rangle) \\ q_1 & \text{if } q = q_0 \text{ and } danger(\langle p_1, p_2 \rangle) \end{cases}$$

$$\Pi(q_1) = \langle q_0, \delta_0 \rangle$$

Note that the conditions of adaptation automata allow us to abstract over the implementation details of both the system and the adaptation function but pinpoint exactly where the adaptation must happen. For instance, in $\mathcal{A}_1$, only the $\star$-transition is enabled in q_2, in contrast with q_0 and q_1 which enable all non-adaptive transitions.

5 REFINEMENT-BASED VERIFICATION

Here we give an encoding of SAA into CSP and use the FDR verifier to prove such policies (safety requirements) in our examples using FDR's trace model. We also use the encoding to reason about functional requirements using the CSP failure model, as well as the failure-divergence model. As we show in this section, our example systems do indeed satisfy the safety requirements, and the functional requirements under the failure model. However they do not satisfy the functional requirements with respect to the failure-divergence model. This is because our systems have inherent internal divergences. In the vehicle system, for example, the vehicles may move over the same positions without exploring the entire search space.

5.1 CSP and FDR

We first briefly overview the structure of CSP [21] processes:

$$P, Q ::= a \rightarrow P \mid P \,\Box\, Q \mid P \underset{A}{\parallel} Q \mid SKIP \mid STOP \mid \text{if } b \text{ then } P \text{ else } P \mid P[[a/b]] \mid P \,\triangle_A\, Q \mid P \backslash A$$

Here $a \rightarrow P$ means that the process P is guarded by action a. CSP allows us to pattern-match a. External (deterministic) choice $P \,\Box\, Q$ allows the environment to choose between synchronising with P or Q. The processes if b then P else P, $P \backslash A$ and $P[[a/b]]$ represent conditional, action hiding and process renaming from b to a respectively. *SKIP* and *STOP* model successful termination and deadlock, respectively. Finally, interleaving $P \underset{A}{\parallel} Q$ requires P and Q to synchronise on actions in the set A but requires no synchronisation on events not in A and $P \,\triangle_A\, Q$ is a hybrid of interrupt and interleaving. As with interrupt, any event that P performs does not resolve the operator. If Q ever performs a visible event that is not in A then this resolves the choice and the process behaves as Q.

The three main semantic models of CSP are trace (T), stable failure (F) and failure-divergence (FD) models. In the trace model, $P \sqsubseteq_{\mathrm{T}} Q$ denotes that $traces(Q) \subseteq traces(P)$. This is useful for specifying safety properties, i.e., all the traces of implementation Q are in the traces of the specification P. In the stable failure model, we compare the set of failures: $failures(Q) \subseteq failures(P)$. A failure is the possibility to refuse a set of actions after performing a trace. In this model, we can define liveliness properties, by showing that if an action is not refused (i.e., it eventually happens) in the specification P, then it is also not refused by implementation Q. Finally, in the failure-divergence model, we further compare the set of diverging processes: $failures(Q) \cup divergence(Q) \subseteq failures(P) \cup divergence(P)$. Here a divergence is a trace that can lead to an infinite loop, thus refusing to perform any observable transition after the loop.

FDR [16] is an automatic refinement tool for machine readable CSP - CSP_M. [34] We can check that an implementation refines the specification according to one of the semantic models through assertions `assert Spec [M= Impl` where `M` ∈ {`T, F, FD`} We can also verify that the implementation is deterministic, deadlock free and livelock free (divergence free) according to one of the semantic models `assert Impl :[deadlock free [M]]`.

5.2 Translation to CSP

Any SAA

$$\mathcal{M} = \langle Q, \Sigma, \Delta, q_0, \delta_0, \Pi \rangle$$

with finite Δ can be translated into CSP_M by following the following steps:

(1) We define a data-type $\widehat{Q}$ representing the set of states Q. We write $\widehat{q}$ for the translated state q.

(2) We define events to represent Σ . We also define low-level adaptation events $\{\star\langle\widehat{q}\rangle \mid q \in Q\}$; where we attach the state to the adaptation event. For instance $\star\langle\widehat{q_0}\rangle$ represents $\Pi(q_0)$.
(3) We translate a transition function $\delta \in \Delta$ as a parametric process taking the state and performing the next transition from that state as defined by δ, i.e.,

$$\widehat{\delta}(\widehat{q}) = \Box\{e \to \widehat{\delta}(\widehat{q'}) \mid e \in \Sigma \text{ and } \delta(q,e) = q'\}$$

The SAA semantics allows us to perform a $\star$-transition if q is an adaptable state ($\Pi(q)$ is defined). For each adaptable state, we add the $\star$-transition to the choice of events that can be performed,

$$\widehat{\delta}(\widehat{q}) = \star\langle\widehat{q}\rangle \to STOP \quad \text{if } \Pi(q) = \langle\delta, q'\rangle$$
$$\Box\{e \to \widehat{\delta}(\widehat{q'}) \mid e \in \Sigma \text{ and } \delta(q,e) = q'\}$$

(4) Adaptation is the replacement of one transition function by another. In the translation this is encoded as the interruption of a process to initiate another process. In our model, adaptation is defined with respect to the state at the time of the adaptation. We utilise the interrupt construct in CSP to implement the adaptation functionality. Recall we communicate the state with an $\star$-transition. Thus, Π needs to evolve to the correct process according to that state. Intuitively,

$$\widehat{\Pi} = \Box \begin{cases} \star\langle\widehat{q_1}\rangle \to \widehat{\delta}'_1(\widehat{q'_1}) & \text{if } \Pi(q_1) = \langle\delta'_1, q'_1\rangle \\ \star\langle\widehat{q_2}\rangle \to \widehat{\delta}'_2(\widehat{q'_2}) & \text{if } \Pi(q_2) = \langle\delta'_2, q'_2\rangle \\ \quad\vdots \end{cases}$$

Assuming $P = (e \to P') \,\Box\, (\star\langle q_1\rangle \to STOP)$. In the process $P \,\triangle_{\star\langle\_\rangle}\, \widehat{\Pi}$, P can evolve though the event e, but as soon as the transition $\star\langle q_1\rangle$ is taken, this event synchronises with one of the events in $\widehat{\Pi}$ and the process $\widehat{\delta}'_1(\widehat{q'_1})$ is initiated instead of P.
Note that the new processes P_1 may need to adapt as well so we recursively include the interrupt construct in $\widehat{\Pi}$,

$$\widehat{\Pi} = \Box\left\{\star\langle\widehat{q}\rangle \to (\widehat{\delta}(\widehat{q'}) \,\triangle_{\star\langle\_\rangle}\, \widehat{\Pi}) \;\middle|\; \begin{array}{c}\Pi(q) = \langle\delta, q'\rangle \\ \text{and } q \in Q_{\mathcal{M}}\end{array}\right\}$$

(5) The final step is to initialise the SAA $\mathcal{M}$ to the initial configuration $\delta_0(q_0)$. Recall that this process can be interrupted by an adaptation as explained in item 4. We define $\mathcal{M}$ as

$$\widehat{\mathcal{M}} = \left(\widehat{\delta_0}(\widehat{q_0}) \,\triangle_{\star\langle\_\rangle}\, \widehat{\Pi}\right)[[\star/\star\langle\widehat{q}\rangle \mid q \in Q]]$$

In our model an SA system is defined as the composition of the system and an adaptation automaton. An adaptation automaton is an SAA and hence can be translated to a CSP process using the technique explained above. The intersection of the two SAA, $\mathcal{A} \cap \mathcal{M}$, can be implemented using the CSP parallel composition construct.

$$\widehat{\mathcal{A}} \underset{Events}{\parallel} \widehat{\mathcal{M}}$$

Because we add state information to all the $\star$-transitions, local to the SAA, we strip such information before composing the two SAA together.

Theorem 5.1. *Consider a SAA $\mathcal{M} = \langle Q, \Sigma, \Delta, q_0, \delta_0, \Pi\rangle$, let $\sigma = [[\star/\star\langle\widehat{q}\rangle \mid q \in Q]]$, then for all $q \in Q$, $\delta \in \Delta$,*

(1) $\langle q, \delta\rangle \xrightarrow{\alpha} \langle q', \delta'\rangle$ implies
$\left(\widehat{\delta}(\widehat{q}) \,\triangle_{\{\star\}}\, \widehat{\Pi}\right)\sigma \xrightarrow{\alpha} \left(\widehat{\delta'}(\widehat{q'}) \,\triangle_{\{\star\}}\, \widehat{\Pi}\right)\sigma$

(2) $\left(\widehat{\delta}(\widehat{q}) \,\triangle_{\{\star\}}\, \widehat{\Pi}\right)\sigma \xrightarrow{\alpha} P$ implies
$P = \left(\widehat{\delta'}(\widehat{q'}) \,\triangle_{\{\star\}}\, \widehat{\Pi}\right)\sigma$ and $\langle q, \delta\rangle \xrightarrow{\alpha} \langle q', \delta'\rangle$

Proof. By induction on the transitions. □

This theorem shows that the translation is a strong bisimulation. Therefore the trace, failure, and failure-divergence models of FDR apply to SAA through the translation.

Example 5.2. Recall the system in section 2, modelled as an SAA in example 4.2. We show the CSP representation of the example with two vehicles.

(1) First we define the set of states $\widehat{Q} = \{\langle p_1, p_2\rangle \mid p_1, p_2 \in Loc\}$
(2) We define the set of events as Σ and the low-level adaptation events which incorporate the system state:

$$\widehat{\Sigma} = \{goto.m_1.m_2 \mid m_1, m_2 \in Loc\} \cup \{\star\langle p_1, p_2\rangle \mid p_1, p_2 \in Loc\}$$

(3) We have two types of transition functions: the initial transition function δ allowing all movements and $\delta_{(m,n)}$ which initially allows one transition (coordinator's command) and then any movement

$$\widehat{\delta_0}(\widehat{q}) = \Box \begin{cases} goto.m_1.m_2 \to \widehat{\delta_0}(\widehat{y}) & \delta_0(q, goto.m_1.m_2) = y \\ \star\langle\widehat{q}\rangle \to STOP & \Pi(q) = \text{ is defined} \end{cases}$$

$$\widehat{\delta}_{m\to n}(\widehat{q}) =$$
$$\Box \begin{cases} goto.m_1.m_2 \to \widehat{\delta}_{m\to n}(\widehat{y}) & \delta_{m\to n}(q, goto.m_1.m_2) = y \\ \star\langle\widehat{q}\rangle \to STOP & \Pi(q) \text{ is defined} \end{cases}$$

(4) The adaptation function Π is implemented as

$$\widehat{\Pi} = \Box\left\{\star\langle p_1, p_2\rangle \to (P \,\triangle_{\star\langle\_\rangle}\, \widehat{\Pi}) \;\middle|\; \begin{array}{c}\Pi(\langle p_1, p_2\rangle) = \langle\delta_{m\to n}, \langle p_1, p_2\rangle\rangle \\ \text{and } P = \widehat{\delta}_{m\to n}(\langle p_1, p_2\rangle)\end{array}\right\}$$

(5) With the above, we implement $\widehat{\mathcal{M}}$ in CSP as

$$\widehat{\delta_0}(\langle l_1, l_2\rangle) \,\triangle_{\star\langle\_\rangle}\, \widehat{\Pi}$$

We translate also the two adaptation automata presented in example 4.6. The adaptation automaton $\mathcal{A}_1$ requires adaptation every two steps. We model $\mathcal{A}_1$ in CSP as shown below,

$$\widehat{Q}_{\mathcal{A}_1} = \{0..2\}$$
$$\widehat{\Sigma} = \{goto.m_1.m_2 \mid m_1, m_2 \in Loc\} \cup \{\star\langle x\rangle \mid x \in 0..2\}$$
$$\widehat{\delta}_{\mathcal{A}_1}(x) = \text{if } x < 2$$
$$\text{then } \Box\{goto.m_1.m_2 \to \widehat{\delta}_{\mathcal{A}_1}(x+1) \mid m_1, m_2 \in Loc\}$$
$$\text{else } \star\langle 2\rangle \to STOP$$
$$\widehat{\Pi} = \star\langle 2\rangle \to (\widehat{\delta}_{\mathcal{A}_1}(0) \,\triangle_{\star\langle 2\rangle}\, \widehat{\Pi})$$

Similarly, the trigger-based adaptation automaton $\mathcal{A}_2$ requires adaptation when the vehicles' locations satisfy the predicate *danger*. We

model $\mathcal{A}_2$ as

$$\widehat{Q}_{\mathcal{A}_2} = \{0,1\} \quad \widehat{\Sigma} = \{goto.m_1.m_2 \mid m_1, m_2 \in Loc\} \cup \{\star\langle 0\rangle, \star\langle 1\rangle\}$$

$$\widehat{\delta}_{\mathcal{A}_2}(0) = \Box \begin{cases} goto.m_1.m_2 \to \widehat{\delta}_{\mathcal{A}_2}(0) & \text{if } m_1, m_2 \in Loc, safe(m_1, m_2) \\ goto.m_1.m_2 \to \widehat{\delta}_{\mathcal{A}_2}(1) & \text{if } m_1, m_2 \in Loc, danger(m_1, m_2) \end{cases}$$

$$\widehat{\delta}_{\mathcal{A}_2}(1) = \star\langle 1\rangle \to STOP$$

$$\widehat{\Pi} = \star\langle 1\rangle \to (\widehat{\delta}_{\mathcal{A}_2}(0) \triangle_{\star\langle 1\rangle} \widehat{\Pi})$$

Finally, $\widehat{\mathcal{A}_1 \cap \mathcal{M}}$ is

$$\texttt{Impl} = \widehat{\delta}_{\mathcal{A}_1}(0)[^{\star}/_{\star\langle \widehat{q}\rangle} \mid q \in Q_{\mathcal{A}_1}] \underset{Events}{\|} \widehat{\mathcal{M}}[^{\star}/_{\star\langle \widehat{q}\rangle} \mid q \in Q_{\mathcal{M}}]$$

Note that $\widehat{\mathcal{A}_2 \cap \mathcal{M}}$ is defined analogously by replacing the adaptation automaton only,

$$\texttt{Impl2} = \widehat{\delta}_{\mathcal{A}_2}(0)[^{\star}/_{\star\langle \widehat{q}\rangle} \mid q \in Q_{\mathcal{A}_2}] \underset{Events}{\|} \widehat{\mathcal{M}}[^{\star}/_{\star\langle \widehat{q}\rangle} \mid q \in Q_{\mathcal{M}}]$$

Properties Proven using FDR. Using FDR, we verify that the system with both adaptation policies satisfy the requirements Req. 1, Req. 2 and Req. 3 from section 2 by proving that the implementations `Impl` and `Impl2` refine the following three specification processes. The safety requirement Req. 1 assert that vehicles never collide or stated differently no two vehicles go to the same location. This can be specified as

```
Req1 = goto?x: Int?y: {y | y<-Int, d(x,y) > 0} -> Req1
```

Similarly, we verify that all vehicles remain within the search space (Req. 2) by showing the implementations refines a specification process `Req2` which recursively accepts any $goto.m_1.m_2$ for m_1 and m_2 within the search space. We assume `Loc` to be the set of all locations in the search space.

```
Req2 = goto?_ : Loc ?_ : Loc -> Req2
```

We use the trace-semantic model to verify safety properties by checking that any derivable trace in the implementation is also derivable in the Specification. In particular, through FDR we can verify the following assertion

```
assert Req1 [T= Impl
```

Note that we can verify the second implementation by either doing similar assertions or alternatively show that the implementations are trace equivalent up-to the adaptation automaton. We verify that if we hide the $\star$ transitions, the implementation are trace equivalent, i.e.,[2]

```
assert Impl2 |\{*} [T= Impl |\{*}
assert Impl |\{*} [T= Impl2 |\{*}
```

For the last requirement Req. 3, we want to verify that all positions in the search area are eventually visited. Since this is a liveness property, trace inclusion is not sufficient. We verify Req. 3 by running the implementation in parallel with a test and check that all possible paths can be extended to pass the test. For each location $l \in$ `Locs`, we define a test that broadcasts a `success` event, signalling that the test passed, when the location l has been visited by a vehicle:

```
T(l) = goto?x?y -> if d(x,p) == 0 or d(y,p) == 0
                   then success -> RUN({|goto|})
                   else T(l)
```

[2] `P |\A` is equivalent to `P \ diff(Events,A)` where `P` performs only events not in `A`

We run all the tests in parallel with the implementation and hide all the events except the `success` events,

```
Tests =
(([|{|goto|}|] l:Locs @ T(l)) [|{|goto|}|] Impl) |\{success}
```

The requirement is satisfied if all the tests pass or stated alternatively the number of success events is the same as the number of locations.

```
Count(n) = n > 0 & success -> Count(n-1)
           n == 0 & ok -> STOP
```

In the assertion, we check that the `ok` event is never refused. The verification ensures that from any path we can eventually broadcast the `ok` event,

```
assert ok->STOP [F= (Count(n) [{success}] Tests) |\{ok}
```

Note that because the vehicles randomly choose the next goto position, the implementation may diverge and running the assertion using the failure-divergence semantic model would rightfully fail. For instance, if the vehicles transition indefinitely between the same pair of locations, such trace can be extended to pass the tests but the test never terminates.

6 EXECUTION MONITORS

Our last result is a study on the expressiveness of SAA by translating SAAs to standard automata. We show that SAA, although dynamically change the transition function, do not alter the model of computation of traditional automata. We do this by translating SAAs to Execution Monitors (EM) [35]. Execution Monitors are automata accepting a prefix-closed set of traces (the monitored traces) which is co-recursively enumerable. This result shows that SAA are unlike other dynamic automata, such as self-modifying finite automata, which are more powerful than standard finite automata [33]. Moreover, since EM-enforceable properties correspond to *safety properties* [35], it follows that adaptation in our framework can enforce all such properties.

Here, we prove that SAA have the same power as EM by providing a bidirectional translation between the two models. This means that adding self-adaptation to automata, as we do in SAA, does not change the computational model. As shown in previous work for self-modifiable automata [33], this is not always the case.

Execution Monitors is a specific sub-class of Büchi automata. An EM $\mathcal{M}$ is formally defined as a tuple $\mathcal{M} = \langle Q, \Sigma, q_0, \delta\rangle$ where

- Q is a potentially infinite set of states
- Σ is a potentially infinite set of actions
- $q_0 \in Q$ is the initial state
- $\delta : Q \times \Sigma \rightharpoonup Q$ is a (partial) transition function.

We write $q \xrightarrow[\delta]{a} q'$ to represent a single transition $\delta(q, a) = q'$, and $q_0 \xrightarrow[\delta]{t}$ to denote the transitive application of δ, where t ranges over a potentially infinite sequence of symbols $a_0, a_1, \ldots$; i.e., $q_0 \xrightarrow[\delta]{a_0} q_1 \xrightarrow[\delta]{a_1} \ldots$.

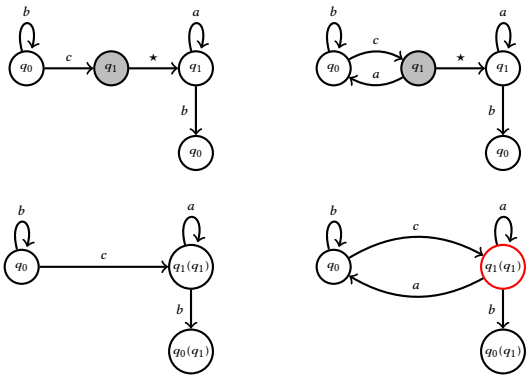

Figure 3: The translation of an SAA to deterministic EM

6.1 Translating Self Adaptive Automata into Execution Monitors

Intuitively, the transformation is a union of all instances of the SAA obtained by adaptations where the ⋆-transitions are forced whenever they are enabled.

The resulting automaton is the combination of all transition functions $\delta \in \Delta$ linked by Π. The transformed states $q_0(q_1)$ give us enough information to map back to the original state in SAA, in this case $q_0(q_1)$ asserts that we are in state q_0 where the last adaptation was on state q_1.

In fig. 3 we present two examples of this translation. The SAA at the top left is translated to the EM at the bottom left of the figure. Note that in this EM we force and hide the ⋆-transition, when enabled (grey state). The SAA at the top right of fig. 3 is not an adaptation automaton because q_1 has both a ⋆- and an a-outgoing transition. Through the translation, the resulting automaton is not deterministic (state $q_1(q_1)$ has two outgoing transitions), and thus it is not an EM.

We show that for an adaptation automaton the translation accepts the same set of traces, modulo the ⋆-events. For a trace t, we let t^- be the trace t stripped of the ⋆-events.

Definition 6.1. For an SAA $\mathcal{M} = \langle Q_{\mathcal{A}}, \Sigma_\star, \Delta, q_0, \delta_0, \prod \rangle$, we write $[\![\mathcal{M}]\!]$ for the EM $\langle Q, \Sigma, q_0, \delta \rangle$ for which

$$Q = Q_{\mathcal{A}} \uplus \{ q(p) \mid q, p \in Q_{\mathcal{A}} \}$$

$$\delta(q,a) = \begin{cases} y'(y) & \text{if } q = x(p),\ \Pi(x) = \langle y, \delta' \rangle \text{ and } y' = \delta'(y,a) \\ y'(p) & \text{else if } q = x(p),\ \Pi(p) = \langle y, \delta' \rangle \text{ and } y' = \delta'(x,a) \\ y'(q) & \text{else if } q \in Q_{\mathcal{A}},\quad \Pi(q) = \langle y, \delta' \rangle \text{ and } \delta'(y,a) = y' \\ y' & \text{else if } q \in Q_{\mathcal{A}},\ \delta_0(q,a) = y' \end{cases}$$

The following theorem shows that $[\![\mathcal{M}]\!]$ and $\mathcal{M}$ accept the same set of traces, and that $[\![\mathcal{M}]\!]$ is an *EM*, provided that $\mathcal{M}$ is deterministic.

Theorem 6.2. *Suppose a SAA $\mathcal{M} = \langle Q_{\mathcal{A}}, \Sigma_\star, q_0, \delta_0, \Pi \rangle$ has adaptation automaton and $[\![M]\!] = \langle Q, \Sigma_\star, q_0, \delta \rangle$; then*

(1) $\langle q_0, \delta_0 \rangle \xrightarrow{t}$ *iff* $q_0 \xrightarrow[\delta]{t^-}$

(2) $[\![\mathcal{M}]\!]$ *is deterministic*

(3) $\mathcal{M}$ *is finite implies* $[\![\mathcal{M}]\!]$ *is finite*

7 RELATED WORK

Hierarchical automata verified through model checking have been extensively used to model SA systems [2, 7, 23, 26, 40]. Bruni et al. [7] and Klarl [26] both present a methodology to compositionally model SA systems by decomposing the main system and the main steps inside the adaptation manager. Bruni et al. illustrated their approach by modelling and verifying an SA system using Maude, whereas Klarl define the HELENA adaptation pattern which is systematically translated to Promela that can be verified using SPIN [22]. Jalili et al. [2] define the HPobSAM framework for modelling and verifying SA systems compositionally. The framework decomposes the system into components with well-defined interfaces. Each components consists of an autonomous manager that implements the adaptation pattern. In these models adaptation events are hard-coded in the system automaton. A change in the adaptation pattern requires potentially significant change in the automaton. In contrast, SAAs allow us to model the system automaton by specifying *may*-adapt events at system states where adaptation is possible. Through composition with a separate adaptation automaton the precise adaptation pattern is selected. This allows us to model sophisticated patterns such as time-triggered adaptation patterns. More importantly, by localising the adaptation pattern, we can experiment with different ones without having to change the main system.

Refinement-based verification has also been proposed for reasoning about SA systems [18, 19]. Hachicha et al. [19] define a set of pre-defined architectures for SA systems and discuss how these can be automatically translated to Event-B for verification purposes. Bartels et al. [4] discuss how to simulate dynamic process creation needed for modelling adaptation in CSP. Göthel et al. [18] extend the framework to include time dependencies. Adaptation is only triggered when the current process finishes executing and hence the system is blocked. Processes are all prefixed with a set of guards and adaptation is implemented by configuring the different guards. It is unclear what is the expressive power of these approaches when it comes to encoding sophisticated adaptation patterns. In SAA adaptation patterns are expressed in the same automata formalism as base systems, which we have shown to be as expressive as Execution Monitors. Therefore adaptation patterns in our framework can enforce all *safety properties* [35]. Moreover, these refinement-based approaches do not localise the adaptation pattern, thus making it difficult to change.

Petri Nets have also been proposed to model and verify SA systems [9, 13, 38]. In particular, Zhang et al. [38] explain how adaptation can be modelled as a transition between petri nets. The model has been extended to time-based petri nets by Camilli et al. [8]. In a similar fashion, Cardozo et al. [9] define context petri nets (CPN) to model adaptation as a dynamic reconfiguration of the petri net. They present a systematic translation of CPN to traditional petri nets. Ding et al. [13] discuss an implementation of the adaptation decision process that wires pre-defined petri nets together based on AI. In these abstract models adaptation is a non-deterministic transition that may lead to many states and hard-coded in the system model. Contrary to this, our framework is more deterministic, where adaptation is a function of the current state of the system. Moreover, our use of *may*-adapt and *must*-adapt events

allows us to decouple the adaptation pattern from the description of the main system, allowing for more modularity.

Runtime models, whereby (possibly partial) system models are generated automatically at runtime to guide the adaptation process have been proposed as a method for correctly implementing SA systems [3, 17, 20, 32, 39]. Goldsby et al. [17] utilise the AVIDA tool to automatically generate UML for different target systems. Zhao et al. [39] outline how RT-UMLs can be model-checked at runtime. This enables the adaptation decision process in SA systems to compare different adaptation alternatives and choose an appropriate one according to non-functional requirements. Pasquale et al. [32] and Almorsy et al. [3] both present tools for adapting security measures at runtime within a SA system to optimally satisfy a requirement in a changing environment. Pasquale et al. present SecuriTAS where requirements are modelled as fuzzy casual networks and the system adjust the security measures to optimally satisfy the network and Almorsy et al. [3] present MDSE@R which utilise aspect-oriented programming techniques to inject adaptable security measures in the system. This line of work focuses mostly on strategies for efficiently implementing dynamic adaptation procedures, geared towards implementation. In principle SAA could be used as a framework to verify the correctness of such systems at design time. The adaptation procedure could be encoded in the Π component of our automata, and the states where adaptation happens could be represented modularly as adaptation pattern automata. The latter is crucial for the correctness of these systems.

Process Languages have been used to model and verify SA systems [5, 12, 27, 36]. Debois et al. [12] define the DCR process language, a Turing-complete declarative process language. A practically useful and decidable sub-class of the language is shown to be equivalent to büchi automata. Unlike our work, adaptation in [12] is implemented as guards on events, which means that both the adaptation process and pattern are embedded within the main system. Lochau et al. [27] define DeltaCCS, an extension for CCS where *may*-adaptation events can be modelled in the language. Verification is through a special-purpose model checker. Schroeder et al. [36] present a model where adaptation is implemented by changing well-defined components that satisfy the same assumptions. Bravetti et al. define $\mathcal{E}$-calculus, a higher-order extension of CCS for modelling adaptable processes [6]. The work studies the (un)decidability of verifying safety and liveness properties with sub-variants of the language. Contrary to these works, SAA are designed to encode adaptation patterns with *must*-adaptation events, something which is difficult to achieve in CCS-based languages without losing modularity of the adaptation patterns. Moreover, we show that SAA can be translated to FDR (and we envisage to other established tools), taking advantage of existing and future verification technology. We have also showed that adaptation in SAA corresponds to EMs, and therefore can enforce all safety properties.

Session types and assume-guarantee reasoning have been used to model adaptation in SA systems [5, 36]. Bono et al. [5] discuss how global session types can implement adaptation within a system. This provides a level of modularity for adaptation patterns, similar to our work. However, adaptation is modelled as a fine-grained communication filter rather than behaviour modification. It is unclear what class of properties can be enforced with this technique. We have showed that adaptation in SAA enforces safety properties.

Other Frameworks have been proposed to model SA systems, such as PobSAM [1, 25], ActivFORMS [24] and Rainbow [10, 14]. PobSAM uses the actor-based language REBECA to model SA systems [1, 25]. The model consists of a collection of managers, actors and policies, one of which is the adaptation automaton. The configuration is modelled using multi-sorted algebra and they present an operational semantics based on an LTS. Iftikhar [24] defines ActivFORMS, an executable model based on timed-automata for modelling the adaptation manager. Garlan et al. [14] define the Rainbow framework where adaptation is defined as a constraint violation on the architecture. Tsigkanos et al. [37] use Bigraphical Reactive Systems (BRS) [31] to model SA systems. The authors focus on cyber physical systems where the system can be decomposed according to the physical layout to improve the scalability of the models' verification, and utilise runtime bounded model checking at pre-defined time intervals to implement the adaptation decision process. These are implementation approaches in which adaptation patterns are chosen by the system designer and are embedded in the approaches. SAA provides an abstract framework for verifying at design-time the correctness of such systems and could potentially be applied in these approaches.

8 CONCLUSIONS

We have presented SAA, an automata-based approach for abstract modelling of self-adaptive systems. In our model we decompose a self-adaptive system into three main components: the base system, providing core functionality, the adaptation decision process, specifying how the system adapts, and the adaptation pattern, specifying when the system adapts. This is achieved through the use of *may*-adapt transitions in the automaton of the base system, and *must*-adapt transitions in the automaton of the adaptation pattern. The full system is derived by the composition of these automata. The adaptation decision process is modelled as a separate function invoked at adaptation transitions. This decomposition of the system allows us to explore multiple adaptation patterns without changing the base system and the adaptation decision process. We have also given a translation from SAA to FDR, though which we can verify key properties of self-adaptive systems, and whether different adaptation patterns break these properties. We have also shown that SAA embody a standard notion of computation and adaptation can enforce all safety properties. Although, our examples can be modelled with vanilla finite automata or CSP, modification to the adaptation pattern in those settings would require significant change to the model.

REFERENCES

[1] Bahareh Abolhasanzadeh and Saeed Jalili. Towards modeling and runtime verification of self-organizing systems. *Expert Systems with Applications*, 44(C):230–244, feb 2016.
[2] Bahareh Abolhasanzadeh and Saeed Jalili. Towards modeling and runtime verification of self-organizing systems. *Expert Systems with Applications*, 44(Supplement C):230 – 244, 2016.
[3] Mohamed Almorsy, John Grundy, and Amani S. Ibrahim. MDSE@R: Model-Driven Security Engineering at Runtime. pages 279–295. Springer, Berlin, Heidelberg, 2012.
[4] Björn Bartels and Moritz Kleine. A csp-based framework for the specification, verification, and implementation of adaptive systems. In *Proceedings of the 6th*

International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pages 158–167. ACM, 2011.

[5] Viviana Bono, Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Data-driven adaptation for smart sessions. *Journal of Logical and Algebraic Methods in Programming*, 90(Supplement C):31 – 49, 2017.

[6] Mario Bravetti, Cinzia Di Giusto, Jorge A. Pérez, and Gianluigi Zavattaro. Adaptable processes. In *Proceedings of the Joint 13th IFIP WG 6.1 and 30th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Systems*, FMOODS'11/FORTE'11, pages 90–105, Berlin, Heidelberg, 2011. Springer-Verlag.

[7] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch Lafuente, and Andrea Vandin. Modelling and analyzing adaptive self-assembly strategies with Maude. *Science of Computer Programming*, 99:75–94, 2015.

[8] Matteo Camilli, Angelo Gargantini, and Patrizia Scandurra. Specifying and verifying real-time self-adaptive systems. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 303–313. IEEE, nov 2015.

[9] Nicolas Cardozo, Sebastian Gonzalez, Kim Mens, Ragnhild Van Der Straeten, and Theo DHondt. Modeling and Analyzing Self-Adaptive Systems with Context Petri Nets. In *2013 International Symposium on Theoretical Aspects of Software Engineering*, pages 191–198. IEEE, jul 2013.

[10] S. W. Cheng, D. Garlan, and B. Schmerl. Evaluating the effectiveness of the rainbow self-adaptive system. In *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 132–141, May 2009.

[11] Maxime Cordy, Andreas Classen, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. Model Checking Adaptive Software with Featured Transition Systems. pages 1–29. Springer Berlin Heidelberg, 2013.

[12] Søren Debois, Thomas Hildebrandt, and Tijs Slaats. *Safety, Liveness and Run-Time Refinement for Modular Process-Aware Information Systems with Dynamic Sub Processes*, pages 143–160. Springer International Publishing, Cham, 2015.

[13] Zuohua Ding, Yuan Zhou, and Mengchu Zhou. Modeling Self-Adaptive Software Systems With Learning Petri Nets. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 46(4):483–498, apr 2016.

[14] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, October 2004.

[15] Ilias Gerostathopoulos, Tomas Bures, Petr Hnetynka, Adam Hujecek, Frantisek Plasil, and Dominik Skoda. Strengthening adaptation in cyber-physical systems via meta-adaptation strategies. *ACM Trans. Cyber-Phys. Syst.*, 1(3):13:1–13:25, April 2017.

[16] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. FDR3 — A Modern Refinement Checker for CSP. In Erika ÃĄbrahÃąm and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.

[17] Heather J. Goldsby and Betty H. C. Cheng. Automatically Generating Behavioral Models of Adaptive Systems to Address Uncertainty. In *Model Driven Engineering Languages and Systems*, pages 568–583. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[18] Thomas Göthel, Nils Jähnig, and Simon Seif. *Refinement-Based Modelling and Verification ofÂăDesign Patterns forÂăSelf-adaptive Systems*, pages 157–173. Springer International Publishing, Cham, 2017.

[19] M. Hachicha, R. B. Halima, and A. H. Kacem. Modeling and verifying self-adaptive systems: A refinement approach. In *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 003967–003972, Oct 2016.

[20] Regina Hebig, Holger Giese, and Basil Becker. Making control loops explicit when architecting self-adaptive systems. *Proceedings of the second international workshop on Self-organizing architectures*, pages 21–28, 2010.

[21] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[22] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

[23] M Usman Iftikhar and Danny Weyns. A case study on formal verification of self-adaptive behaviors in a decentralized system. *arXiv preprint arXiv:1208.4635*, 2012.

[24] M. Usman Iftikhar and Danny Weyns. ActivFORMS: active formal models for self-adaptation. In *Proc. of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems - SEAMS 2014*, pages 125–134. ACM Press, 2014.

[25] Narges Khakpour, Saeed Jalili, Carolyn Talcott, Marjan Sirjani, and MohammadReza Mousavi. Formal modeling of evolving self-adaptive systems. *Science of Computer Programming*, 78(1):3–26, nov 2012.

[26] Annabelle Klarl. Engineering Self-Adaptive Systems with the Role-Based Architecture of Helena. In *2015 IEEE 24th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 3–8. IEEE, jun 2015.

[27] Malte Lochau, Stephan Mennicke, Hauke Baller, and Lars Ribbeck. *DeltaCCS: A Core Calculus for Behavioral Change*, pages 320–335. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[28] Frank D Macías-Escrivá, Rodolfo Haber, Raul del Toro, and Vicente Hernandez. Self-adaptive systems: A survey of current approaches, research challenges and applications. *Expert Systems with Applications*, 40(18):7267–7279, December 2013.

[29] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, July 2004.

[30] Emanuela Merelli, Nicola Paoletti, and Luca Tesei. Adaptability checking in complex systems. *Science of Computer Programming*, 115:23–46, 2016.

[31] Robin Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.

[32] Liliana Pasquale, Claudio Menghi, Mazeiar Salehie, Luca Cavallaro, Inah Omoronyia, and Bashar Nuseibeh. SecuriTAS: A Tool for Engineering Adaptive Security. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12*, page 1, New York, New York, USA, nov 2012. ACM Press.

[33] Roy S. Rubinstein and John N. Shutt. Self-modifying finite automata: An introduction. *Information Processing Letters*, 56(4):185–190, nov 1995.

[34] Bryan Scattergood. *The semantics and implementation of machine-readable CSP*. PhD thesis, Citeseer, 1998.

[35] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, February 2000.

[36] Andreas Schroeder, Sebastian S. Bauer, and Martin Wirsing. A contract-based approach to adaptivity. *Journal of Logic and Algebraic Programming*, 80(3-5):180–193, apr 2011.

[37] Christos Tsigkanos, Liliana Pasquale, Carlo Ghezzi, and Bashar Nuseibeh. On the Interplay Between Cyber and Physical Spaces for Adaptive Security. *IEEE Transactions on Dependable and Secure Computing*, PP(99), 2017.

[38] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceeding of the 28th international conference on Software engineering - ICSE '06*, page 371. ACM Press, 2006.

[39] Y. Zhao, S. Oberthür, M. Kardos, and F.J. Rammig. Model-based Runtime Verification Framework for Self-optimizing Systems. *Electronic Notes in Theoretical Computer Science*, 144(4):125–145, may 2006.

[40] Yongwang Zhao, Dianfu Ma, Jing Li, and Zhuqing Li. Model Checking of Adaptive Programs with Mode-extended Linear Temporal Logic. In *2011 Eighth IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems*, pages 40–48. IEEE, apr 2011.