A Case Study on Formal Verification of Self-Adaptive Behaviors in a Decentralized System

M. Usman Iftikhar   Danny Weyns

School of Computer Science, Physics and Mathematics
Linnaeus University, Växjö, Sweden
`usman.iftikhar@lnu.se, danny.weyns@lnu.se`

Self-adaptation is a promising approach to manage the complexity of modern software systems. A self-adaptive system is able to adapt autonomously to internal dynamics and changing conditions in the environment to achieve particular quality goals. Our particular interest is in decentralized self-adaptive systems, in which central control of adaptation is not an option. One important challenge in self-adaptive systems, in particular those with decentralized control of adaptation, is to provide guarantees about the intended runtime qualities. In this paper, we present a case study in which we use model checking to verify behavioral properties of a decentralized self-adaptive system. Concretely, we contribute with a formalized architecture model of a decentralized traffic monitoring system and prove a number of self-adaptation properties for flexibility and robustness. To model the main processes in the system we use timed automata, and for the specification of the required properties we use timed computation tree logic. We use the Uppaal tool to specify the system and verify the flexibility and robustness properties.

1 Introduction

Our society extensively relies on the qualities of software systems, e.g., the reliability of software for media, the performance of software for manufacturing, and the openness of software for enterprise collaborations. However, ensuring the required qualities of software that has to operate in dynamic environments poses severe engineering challenges. Self-adaptation is generally considered as a promising approach to manage the complexity of modern software systems [12, 14, 5, 15]. Self-adaptation enables a system to adapt itself autonomously to internal dynamics and changing conditions in the environment to achieve particular quality goals. It is widely recognized that software architecture provides the right level of abstraction and generality to deal with the challenges of self-adaptation [17, 8, 14]. In particular, the use of an architecture-based approach can provide an appropriate level of abstraction to describe dynamic change in a system, such as the use of components, bindings and composition, rather than at the algorithmic level. Our particular interest is in decentralized self-adaptive systems, in which central control of adaptation is not an option. Examples are large-scale traffic systems, integrated supply chains, and federated cloud infrastructures.

One important challenge in self-adaptive systems, in particular those with decentralized control of adaptation, is to provide guarantees about the required runtime quality properties. In previous research, we have defined formally founded design models for decentralized self-adaptive systems that cover *structural* aspects of self-adaptation [24]. These models support engineers with reasoning about structural properties, such as types and interface relations of different parts of the decentralized system. However, in order to provide guarantees about qualities, we need to complement this work with an approach to validate *behavioral* properties of decentralized self-adaptive systems. The need for research on for-

mal verification of behavioral properties of self-adaptive systems is broadly recognized by the community [16, 25, 21, 15].

This paper reports a first step of our research goal to develop an integrated approach to validate behavioral properties of decentralized self-adaptive systems to guarantee the required qualities [22]. This approach integrates three activities: (1) model checking of the behavior of a self-adaptive system during design, (2) model-based testing of the concrete implementation during development, and (3) runtime diagnosis after system deployment. The key underlying idea of the approach is to enhance validation of qualities by transfering formalization results over different phases of the software life cycle, e.g., model based testing starts with a verified model and a set of required properties and then intends to show that the implementation of the system behaves compliant with this model. The focus of this paper is on the first activity. Concretely, we present a case study of a decentralized traffic monitoring system and use model checking to guarantee a number of self-adaptation properties for flexibility and robustness. With flexibility we refer to the ability of the system to adapt dynamically with changing conditions in the environment, and robustness is the ability of the system to cope autonomously with errors during execution. We model the main system processes with timed automata and specify the required properties using timed computation tree logic (TCTL). We use the Uppaal tool that offers an integrated environment for modeling, simulation and verification, based on automata and a subset of TCTL.

The remainder of this paper is structured as follows. In Section 2, we introduce the traffic monitoring system and explain a number of adaptation scenarios. In Section 3, we give a brief background on formal modeling with Uppaal. Section 4 presents the design model of the traffic monitoring system, and Section 5 explains how we verified key properties and discusses potential uses of the study results both as input for model based testing and as a starting point for the definition of a reusable behavior model for self-adaptive systems. We discuss related work in Section 6, and conclude with a summary and challenges ahead in Section 7.

2   Traffic Monitoring System

Intelligent transportation systems (ITS) is a worldwide initiative to exploit information and communication technology to improve traffic.[1] One of the challenges in this area is effective monitoring of traffic. In [23], we have introduced a monitoring system that provides information about traffic jams. This information can be used to reduce traffic congestion by different types of clients, such as traffic light controllers, driver assistance systems, etc. The main challenges of the system are: (1) inform clients of dynamic changing traffic jams, (2) realize this functionality in a decentralized way, avoiding the bottleneck of a centralized control center, (3) make the system robust to camera failures. Whereas the focus in [23] was on the structural aspects, here we focus on the behavioral aspects of the system' architecture.

The system consists of a set of intelligent cameras, which are distributed along the road. An example of a highway is shown in Fig. 1. Each camera has a limited viewing range and cameras are placed to get an optimal coverage of the highway with a minimum overlap. To realize a decentralized solution, cameras collaborate in organizations: if a traffic jam spans the viewing range of multiple cameras, they form an organization that provides information to clients that have an interest in traffic jams.

Fig. 1 shows two scenarios that require adaption. The first scenario concerns the dynamic adaptation of an organization from T0 to T1, where camera 2 joins the organization of cameras 3 and 4 after it monitors a traffic jam. The second scenario concerns robustness to a silent node failure, i.e., a failure in which a failing camera becomes unresponsive without sending any incorrect data. This scenario is

---

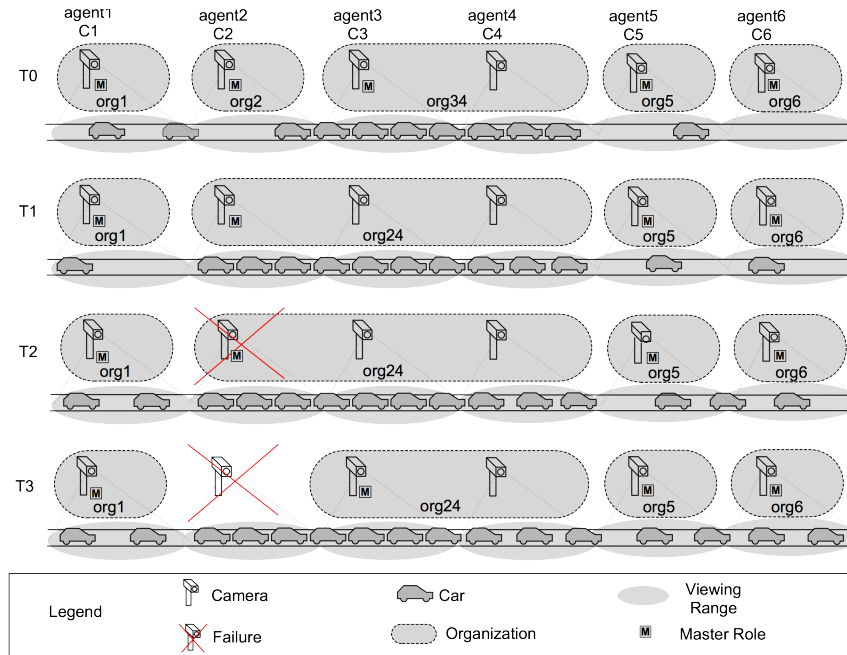[1]http://ec.europa.eu/transport/its/, http://www.its.dot.gov/

Figure 1: Self-healing scenario

shown from T2 to T3, where camera 2 fails. Since there are dependencies between the software running on different cameras (details below), such failures may bring the system in an inconsistent state and disrupt its services. Therefore, the system should be able to restore its services after a failure, although in degraded mode since the traffic state is no longer monitored in the viewing range of the failed camera.

2.1 Dynamic Agent Organizations for Flexibility

Figure 2a shows the primary components of the software deployed on each camera, i.e. the *local camera system*. The *local traffic monitoring system* provides the functionality to detect traffic jams and inform clients. The local traffic monitoring system is conceived as an agent-based system consisting of two components. The *agent* is responsible for monitoring the traffic and collaborating with other agents to report a possible traffic jam to clients. The *organization middleware* offers life cycle management services to set up and maintain organizations. We employ dynamic organizations of agents to support flexibility in the system, that is, agent organizations dynamically adapt themselves with changing traffic conditions. To access the hardware and communication facilities on the camera, the local traffic monitoring system can rely on the services provided by the *distributed communication and host infrastructure*.

In normal traffic conditions, each agent belongs to a single member *organization*. However, when a traffic jam is detected that spans the viewing range of multiple neighboring cameras, organizations on these cameras will merge into one organization. To simplify the management of organizations and interactions with clients, the organizations have a master/slave structure. The master is responsible for managing the dynamics of that organization (merging and splitting) by synchronizing with its slaves and neighboring organizations and reporting traffic jams to clients. Therefore, the master uses the context information provided by its slaves about their local monitored traffic conditions. At T0, the example in Fig. 1 shows four single member organizations, *org1* with *agent1*, *org2* with *agent2*, and similar for *org5*, and *org6*. Furthermore, there is one merged organization, *org34* with *agent3* as master and *agent4*

(a) Primary software components deployed on each camera

(b) Design model of camera and environment (communication channels are omitted)
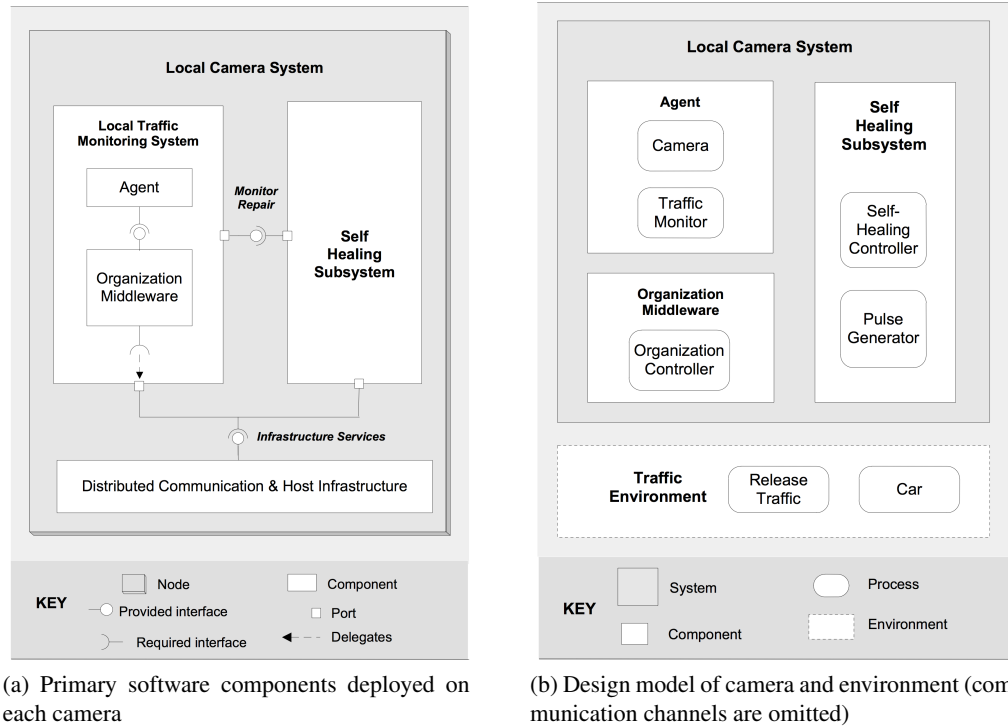
Figure 2: Local Camera System

as slave. At T1, the traffic jam spans the viewing range of cameras 2, 3 and 4. As a result, organizations *org2* and *org34* have merged to form *org24* with *agent2* as master. When the traffic jam resolves, the organization is split dynamically.

2.2   Self-Healing Subsystem for Robustness

To recover from camera failures, a *self-healing subsystem* is added to the local traffic monitoring system, as shown in Fig. 2a. The self-healing subsystem maintains a model of the current dependencies of the components of the local traffic monitoring system with other active cameras. Each working camera is in one of three distinct roles: master of a single member organization, master of an organization with slaves, or slave in an organization. As these roles come with certain responsibilities, each camera is dependent on a particular set of remote cameras in order to function properly: (1) a master of a single agent organization is dependent on its neighboring nodes; (2) a master with slaves is dependent on its slaves and its neighboring nodes; (3) a slave of an organization is dependent on its master and its neighboring nodes.

    To recover from camera failures, the subsystem contains repair actions for failure scenarios in different roles. Examples of actions are: halt the communication with the failed neighboring camera, elect a new master, and exchange the current monitored traffic state with another camera. To detect failures, the self-healing subsystem coordinates with self-healing subsystems on other cameras in the dependency model using a ping-echo mechanism. Cameras send periodically ping messages to dependent cameras and a failure is detected when a camera does not respond with an echo after a certain time. [23] provides a detailed description of the structural architecture of the traffic monitoring system.

3   Uppaal

Model-checking is verifying a given model w.r.t. a formally expressed requirement specification. Uppaal is a model-checking tool for verification of behavioral properties [4]. In Uppaal, a system is modeled as a network of timed automata, called processes. A timed automaton is a finite-state machine extended with clock variables. A clock variable evaluates to a real number, and clocks progress synchronously. It is important to note that fulfilled constraints for the clock values only enable state transitions but do not force them to be taken. A process is an instance of a parameterized template. A template can have local declared variables, functions, and labeled locations. The syntax to declare functions is similar to that of the C language. State of the system is defined by locations of the automata, clocks, and variables values.

Uppaal uses a subset of TCTL for defining requirements, called the query language. The query language consists of state formulae and path formulae. State formulae describe individual states with regular expressions such as $x >= 0$. State formulae can also be used to test whether a process is in a given location, e.g., *Proc.loc*, where *Proc* is a process and *loc* is a location. Path formulae quantify over paths of the model and can be classified into *reachability*, *safety*, and *liveness* properties:

- *Reachability* properties are used to check whether a given state formula ϕ can be satisfied by some reachable state. The syntax for writing this property is $E <> \phi$.

- *Safety* properties are used to verify that "something bad will never happen." There are two path formulae for checking safety properties. $A[] \phi$ expresses that a given state formula ϕ should be true in all reachable states, and $E[] \phi$ means that there should exist a path that is either infinite, or the last state has no outgoing transitions, called maximal path, such that ϕ is always true.

- *Liveness* properties are used to verify that something eventually will hold, which is expressed as $A <> \phi$. The property "whenever ϕ holds, eventually ψ will happen" is stronger and is expressed as $\phi \rightarrow \psi$.

Processes communicate with each other through *channels*. *Binary channels* are declared as *chan x*. The sender *x*! can synchronize with the receiver *x*? through an edge. If there are multiple receivers *x*? then a single receiver will be chosen non-deterministically. The sender *x*! will be blocked if there is no receiver. *Broadcast channels* are declared as *broadcast chan x*. The syntax for sender *x*! and receiver *x*? is the same as for binary channels. However, a broadcast channel sends a signal to all the receivers, and if there is no receiver, the sender will not be blocked. Uppaal also supports arrays of channels. The syntax to declare them is *chan x[N]* or *broadcast chan x[N]*, and sending and receiving signals are specified as *x[id]*! and *x[id]*?. Note that processes cannot pass data through signals. If a process wants to send data to another process then the sender has to put the data in a shared variable before sending a signal and the receiver will get the data from shared variable after receiving the signal.

Uppaal offers a graphical user interface (GUI) and model checking engine. The GUI consists of three parts: the editor, the simulator and the verifier. The editor is used to create the templates. The simulator is similar to a debugger, which can be used to manually run the system, showing running process, their current location and the values of the variables and clocks. The verifier is used to check properties of the model as described above.

4   Model Design in Uppaal

We now discuss the formal design of the traffic monitoring system in Uppaal. We already explained in section 2 that each camera consists of 3 primary components: *Agent*, *Organization Middleware* and *Self-healing Subsystem*. We have designed each of these components as a set of timed automata (templates)

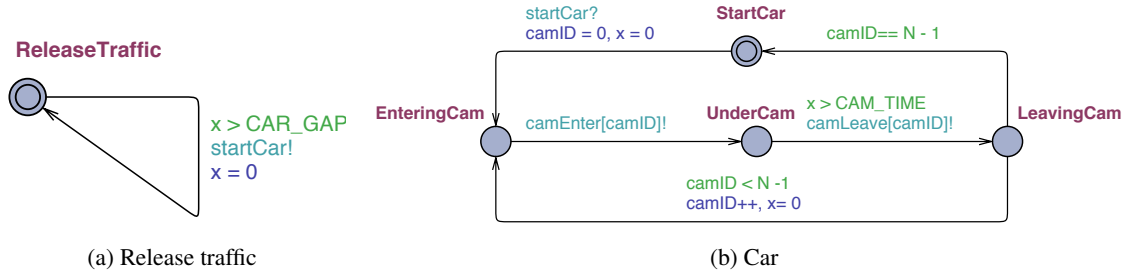(a) Release traffic   (b) Car

Figure 3: Environment processes

that represent abstract processes. Fig. 2b (section 2.1) shows how the processes map to the components of the system. To instantiate a particular system model, each template is instantiated to one or several concrete processes. We use channels to enable processes to communicate within a camera and between cameras. To that end, the id of the receiver camera is used. We start by defining the different templates of the system. Then we explain how the templates are instantiated into a concrete system model.

4.1 Environment Processes

The environment is modeled as two simple timed automata: *Release Traffic* and *Car*.

- *Release Traffic* is an abstract model of the traffic environment. Fig. 3a shows the template. The purpose of traffic release is to feed the system with cars after some non-deterministic time *CAR_GAP*. Variable x is a local clock and whenever its value is greater than *CAR_GAP*, the *StartCar* signal is emitted. Only one instance of the release traffic process will be running all the time.

- *Car* is the abstract model of a car in the environment. Fig. 3b shows the template. *Car* waits for the *startCar* signal from the release traffic process.[2] Once started, the car moves along the subsequent viewing ranges of the cameras. Whenever a car enters/leaves the viewing range of a particular camera it emits a signal. This allows the camera agents to monitor traffic congestion. As in real traffic, the car template will have many running instances, each representing a car in the system.

4.2 Agent Processes

An agent is modeled as two timed automata: *Camera* and *Traffic Monitor*.

- Each *Camera* has four basic states. In normal operation, the camera can be master with no slaves, master of an organization with slaves, or it can be slave. Additionally, the camera can be in the failed state, representing the status of the camera after a silent node failure. Fig. 4a shows the template. There is an instance with a unique id for each camera. Cameras in the master status are responsible for communicating the traffic conditions to clients, but this functionality is not modeled here.

- *Traffic Monitor* keeps track of the actual traffic conditions based on the signals it receives from the cars and determines traffic congestion. It interacts with the local organizational manager to handle organization management. Fig. 4b shows the template. For each camera, one traffic monitoring process instance is running all the time. Whenever a car enters into the viewing range of a camera, the

---

[2]*startCar* is the initial location marked by two circles.
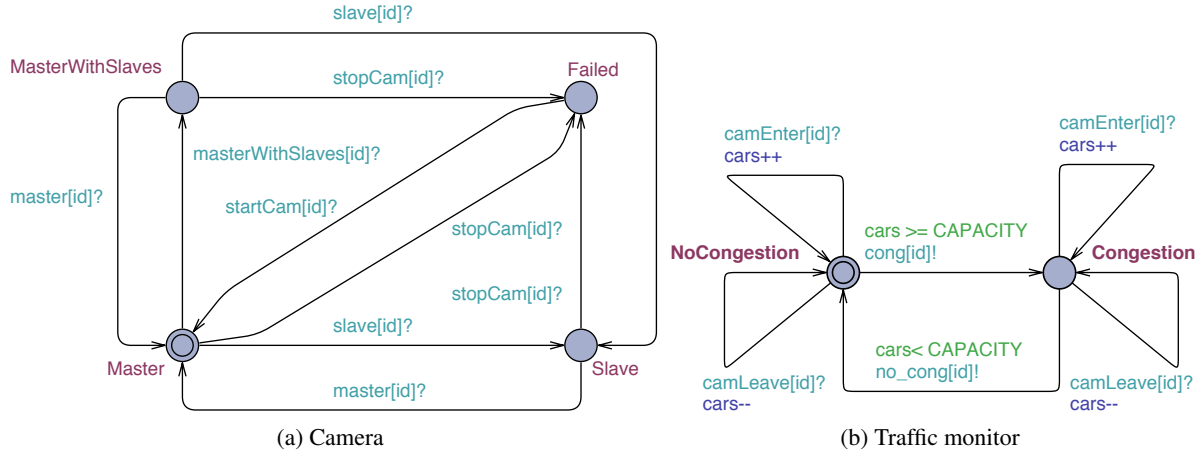
(a) Camera

(b) Traffic monitor

Figure 4: Agent processes

traffic monitor detects the car via the *camEnter* channel. Similarly when a car goes out of the range of a camera, the traffic monitor detects this through the *camLeave* channel. The traffic monitors determines a traffic jam by comparing the total number of cars in its viewing range with the *CAPACITY*. Based on this, the monitor may interact with the organization controller to adapt the organizations.

4.3 Organization Middleware

The organization middleware is modeled as one timed automaton: *Organization Controller*.

- *Organization Controller* is responsible for managing organizations, based on the information it gets from the traffic monitor process of the agent. Fig. 5 shows the template.[3] An organization middleware process runs on each camera. A camera starts as master of a single member organization. When a *congestion* is detected the organization controller sends a *request_org* signal to the neighboring camera in the direction of the traffic flow. Depending on the traffic conditions of the neighboring camera and the current role of the camera, the organizations may be restructured as follows. If traffic is not jammed in the viewing range of the neighbor, organizations are not changed. If traffic is jammed and the neighbor is *Slave* or *MasterWithSlaves*, the camera joins the organization as a slave. If both are masters of single member organizations (*Master*), the camera with the highest id becomes master with slaves of the joined organization (transition *OrgOfferMaster* to *TurningMasterWithSlaves* to *MasterWithSlaves*), while the other will become slave (transition *OrgOfferMaster* to *TurningSlave* to *Slave*). A master with slaves can add and remove slaves dynamically. When no slave remains, the master with slaves becomes master again. Whenever, the role of a camera changes, the organization controller informs the camera process to update its status via signals *slave*[*id*], *master*[*id*], or *masterWithSlaves*[*id*] respectively. If the organization controller receives the *stopCam* signal, it will go to *Failure* state, which represents a silent node failure. The controller will not respond until it is recovered via the *startCam* signal.

---

[3]Committed states are marked with C. These states cannot delay, and the next transition must involve an outgoing edge of at least one of the committed locations.
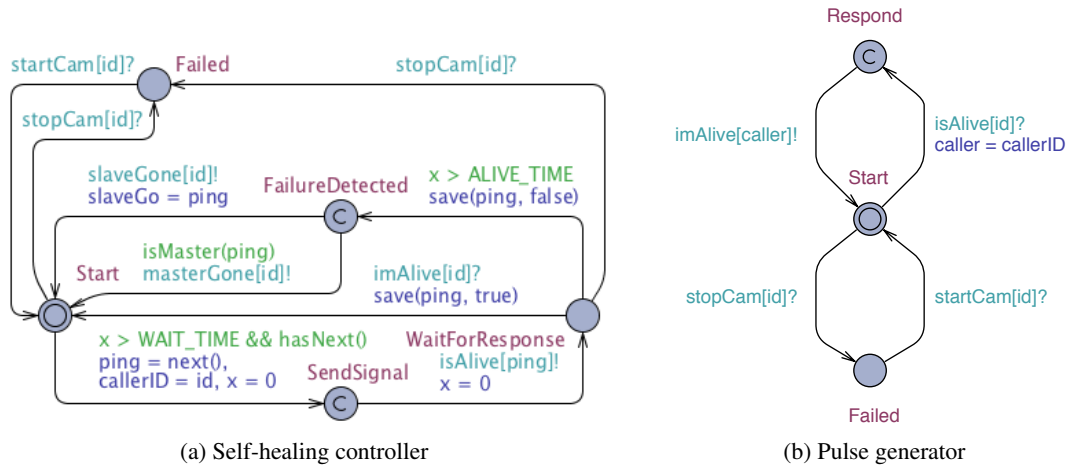
Figure 5: Organization controller

(a) Self-healing controller   (b) Pulse generator

Figure 6: Processes self-healing subsystem

4.4 Self-Healing Subsystem

The self-healing subsystem is modeled as two automata: *Self-Healing Controller* and *Pulse Generator*.

- *Self-Healing Controller* is used to detect failures of other cameras based on a ping-echo mechanism. Fig. 6a shows the template. A self-healing controller process runs on each camera. The self-healing controller sends periodically *isAlive[ping]* signals (based on *WAIT_TIME*) to the self-healing controllers of the dependent cameras. If a camera does not respond in a certain time (*ALIVE_TIME*) it adapts the organizational controller, either by removing a dependency in case a slave failed, or by restructuring the organization in case the master of the organization failed.

- *Pulse Generator* is responsible to respond to the ping signals sent by other cameras to check whether a particular camera is alive or not. Fig. 6b shows the template.

4.5 Definition Model Instance

A concrete system model is defined in Uppaal's system declarations section by listing the processes that have to be composed into a system:

```
system Camera, TrafficMonitor, OrganizationController,
       SelfHealingController, PulseGenerator,
       ReleaseTraffic, Car;
```

To define a concrete model, each template has to be instantiated to concrete processes. The process instances are defined in Uppaal's project declarations section:

```
const int N = 6;              // # Camera
typedef int[0, N-1] cam_id;
...
// Global Constants
const int CAM_TIME = 10;
const int RECOVER_TIME = 500;
...
// Channels
```

```
chan startCar;
chan camEnter[N], camLeave[N], no_cong[N];
...
chan reqTrafficJam, repTrafficJam[N];
broadcast chan request_org[N], change_master[N], congestion[N];
...
```

The declaration defines a setup with 6 camera systems. As each local camera system has 5 processes (see Fig. 2), we will have 30 processes running, plus the environment processes. Unique identifiers are used to identify related processes to work together. We have used an array of channels to enable processes to communicate within each camera and between cameras. If a process has to communicate with another process of the same camera it uses the id of the camera. If a process wants to communicate with a process of another camera it will use the id of the other camera.

5   Model Checking

Based on the formal design of the traffic monitoring system, we can now check properties of the traffic monitoring system by Uppaal's verifier. We have divided the properties in 3 groups, respectively system invariants, correctness of dynamic organization adaptations, and correctness of robustness to silent node failures. We start by defining the properties. Then, we discuss the design and verification process.

5.1   System Invariants

The following properties should hold:

- All cameras cannot be slave at the same time (the system would not provide its function, that is, there are no masters that inform clients about traffic congestion):

```
I1:   A[] not forall(i: cam_id) Camera(i).Slave
```

- The system should not be in deadlock at any time. A deadlock may for example occur when the self-healing controllers are waiting on each other for responses to ping messages and none of them is able to send a response. Such situations should obviously be avoided.

```
I2:   A[] not deadlock
```

Checking for deadlock is directly supported by Uppaal.

5.2   Flexibility Properties

To guarantee that the system adapts itself dynamically to the changing traffic conditions, we define properties that allow verification of correct merges of organizations, such as the first scenario described in Fig. 1. In this scenario, camera 1 merges with the existing organization of camera 2 and 3. In the resulting joined organization at T1, camera 1 is master and the other cameras are slaves. The properties for verifying correct merging are formulated as follows:

- When the organization controller of a camera detects congestion (thus being master of a single member organization), then the camera merges with the neighboring organization in the direction of the traffic flow if this organization is detecting jammed traffic. Formally, we distinguish between three properties according to the current role of the neighboring camera, i.e., master with slaves, slave, and master of a single member organization. These properties are defined as follows:

```
F1:  A<> forall(n : cam_id)
        OrganizationController(n).CongestionDetected
            && Camera(n+1).MasterWithSlaves
        imply
            Camera(n).MasterWithSlaves
            && camera[n].slaves[n+1]

F2:  A<> forall(n : cam_id) forall(x : cam_id)
        OrganizationController(n).CongestionDetected
            && Camera(n+1).Slave
            && camera[x].slaves[n+1]
        imply
            Camera(x).MasterWithSlaves
            && camera[x].slaves[n]

F3:  A<> forall(n : cam_id)
        OrganizationController(n).CongestionDetected
            && Camera(n+1).Master
            && OrganizationController(n+1).CongestionDetected
        imply
            Camera(n).MasterWithSlaves
            && camera[n].slaves[n+1] ||
            Camera(n+1).MasterWithSlaves
            && camera[n+1].slaves[n]
```

In case the neighboring camera is master with slaves (property F1), the camera joins the organization and becomes master. In case the neighboring camera is slave (property F2), the camera joins the neighboring organization as slave. In case both are masters of single member organizations (property F3), they merge and one of them becomes master.

5.3   Robustness Properties

To guarantee that the system recovers from a silent node failure, we define properties that allow verification of correct adaptations of organizations, such as the first scenario described in Fig. 1. In this scenario camera 2 fails at T3, which is the master of an organization with two slaves. Subsequently, the slaves elect a new master and the system recovers from the failure. To introduce failing cameras in the system, we modeled a virtual environment template that allows us to create sequences of traffic conditions, such as those described in Fig. 1. Fig. 7 shows this template. We focus here on properties to verify robustness of a failure of a camera in the role of master with slaves. The properties to verify robustness for failing cameras in other roles are similar.

The scenario of Fig. 1 is defined in the template's declarations section as follows:

```
clock x;
cam_id id = 0;

bool isTrafficJam(cam_id id){
    if (id == 0) return false; if (id == 1) return true;
    if (id == 2) return true; if (id == 3) return true;
    if (id == 4) return false; if (id == 5) return false;
    return false;
}
```
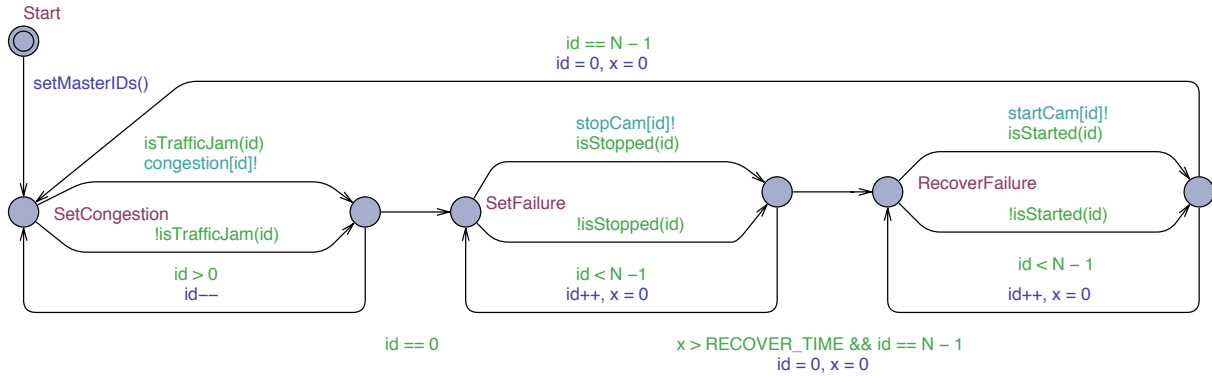
Figure 7: Environment template to inject camera failures

```
bool isStopped(cam_id id){
    if (id == 0) return false; if (id == 1) return true;
    if (id == 2) return false; if (id == 3) return false;
    if (id == 4) return false; if (id == 5) return false;
    return false;
}

bool isStarted(cam_id id){
...

void setMasterIDs(){
    for (i : cam_id) camera[i].m_cam = i;
    id = N-1;
}
```

Verifying robustness of a failing master consists of three parts:

1. When the master camera fails, eventually, the self-healing controllers of the slaves detect the failure,

2. When the slaves have detected the failure, the organization controllers of the slaves will form a new organization,

3. Finally, the cameras will continue their function and monitor traffic jams.

To verify the correct recovering of the organization we defined the following properties:

• When camera 2 fails then eventually the self-healing controllers of the slaves detect the failure.

```
R1:  A<> SelfHealingController(2).Failed
        imply
          SelfHealingController(3).FailureDetected
          && SelfHealingController(4).FailureDetected
```

• If organization controller 2 fails and organization controller 3 or 4 has detected this (by switching to master), then eventually the organization controller of either camera 3 or 4 switches to master with slave, and the other camera becomes slave.

```
R2:  OrganizationController(2).Failed &&
      ((OrganizationController(3).Master
```

```
         && OrganizationController(4).Slave) ||
      (OrganizationController(4).Master
         && OrganizationController(3).Slave))
   -->
    OrganizationController(2).Failed &&
     ((OrganizationController(3).MasterWithSlaves
         && camera[3].slaves[4]) ||
      (OrganizationController(4).MasterWithSlaves
         && camera[4].slaves[3]))
```

- When camera 2 fails then eventually camera 3 and 4 will continue monitoring a traffic jam as a correct organization.

```
R3: A<> Camera(2).Failed
     imply
       ((Camera(3).MasterWithSlaves
           && camera[3].slaves[4]) ||
        (Camera(4).MasterWithSlaves
           && camera[4].slaves[3]))
```

Finally, we verify whether neighbor relations[4] are correctly restored after a failure:

```
R4:  A<> forall(n: cam_id) forall(x:cam_id)
        (Camera(n).Failed && Camera(x).getLeftNeighbour() == n
        imply
           SelfHealingController(x).FailureDetected
           && Camera(x).getLeftNeighbour() == n - 1)
```

We defined a similar rule for neighbors on the right hand side.

5.4 Design and Verification Process

For the design of the models of the managed system (Camera, Traffic Monitor, and Organization Controller) and the managing system (Self-Healing Controller and Pulse Generator), we used Uppaal's simulator to check and correct the design. In this stage, we used the environment models (Release Traffic and Car) as shown in Fig. 3 to test the different models. Then we defined the system invariants and the flexibility and robustness properties. To verify these properties, we designed the virtual environment template as shown in Fig. 7. This restricted environment definition was needed, as verification of an arbitrary environment with randomly injected camera failures suffers from the state space problem. On the other hand, the virtual environment has the advantage that we could define the scenarios we wanted to verify with different number of cameras and increasing complexity.

To give an idea of the time required for verification, we verified the different properties for an increasing number of cameras and measured the verification time. Verification for checking that the system works properly (invariant I1) increases from 1.2 sec for 6 cameras to 197.3 sec for 60 cameras, and from 7.2 to 1330.6 sec for checking deadlock (I2). We also measured the verification time for flexibility and robustness properties. Fig. 8 shows the results for properties F2 and R3. The figure shows that the verification time for these properties grows quasi linear with the number of cameras.

By activating the "diagnostic trace" option, Uppaal can show counter examples in the simulator environment when a property is violated. This option allows to analyze the system's behavior with respect to the property and identify possible design errors. At the end of the design of the traffic monitoring system,

---

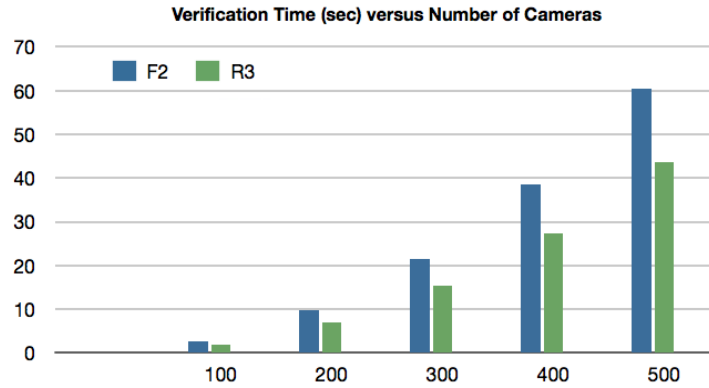[4]For the scenario with 6 cameras, we assigned the first camera as the neighbor of the last and vice versa.

Figure 8: Verification time for increasing number of cameras

all properties were satisfied which was a requirement as they are defined as system requirements. The Uppaal models of the traffic monitoring system with a prototype Java implementation of the system is available for download via: http://homepage.lnu.se/staff/daweaa/TrafficCaseUppaal.html.

5.5   Discussion

In this section, we discuss two topics. We start with explaining how the work presented in this paper fits in the integrated approach for validating quality properties of self-adaptive systems. Next, we discuss a reusable behavior model for self-adaptive systems that we derived from our study.

As explained in the introduction, the work presented in this paper fits in an integrated approach that aims to exploit formalization results of model checking to support model-based testing of concrete implementations and runtime diagnosis after system deployment. To that end, it is our goal to employ the verified models to test the prototype implementation using model-based testing. The goal of model based testing is to show that the implementation of the system behaves compliant with this model. Model based testing uses a concise behavioral model of the system under test, and automatically generates test cases from the model. As the focus of model-based testing so far has mainly been on functional correctness of software systems [20], and self-adaptation if primarily concerned with quality properties, we face several challenges here. A key challenges is to identify the required models to support model-based testing of quality properties. We belief that environment models are a sine qua non for model based testing of runtime qualities, which is central to self-adaptation. In our case study, it is the environment model that specifies the failure events that have to be tested. An explicit model of the environment allows an engineer to precisely specify the failure scenarios of interest and the conditions under which the failures happens. For example, in the scenario shown in Fig. 7 a camera failure is generated after traffic is congested, which allows to test the correctness of the system when one of the cameras of an organization that monitors a traffic jam fails. Another challenge is to define proper test selection. As exhaustive testing of realistic systems is typically not feasible, the tester needs to steer test selection. As an example, to test the self-healing scenario described in Fig. 1, we could mark the RecoverFailure state in the automaton shown in Fig. 7 as a success state. We can then formulate a reachability property to check whether the system will always reach the recover failure state after the camera has failed. This property can be issued to the model checker to test whether the implementation conforms to the model with respect to this property.

Finally, from our experiences with the case study, we derived an interesting model that maps the

different types of behaviors of self-adaptive systems to zones of the state space. Fig. 9 shows an overview of the model.
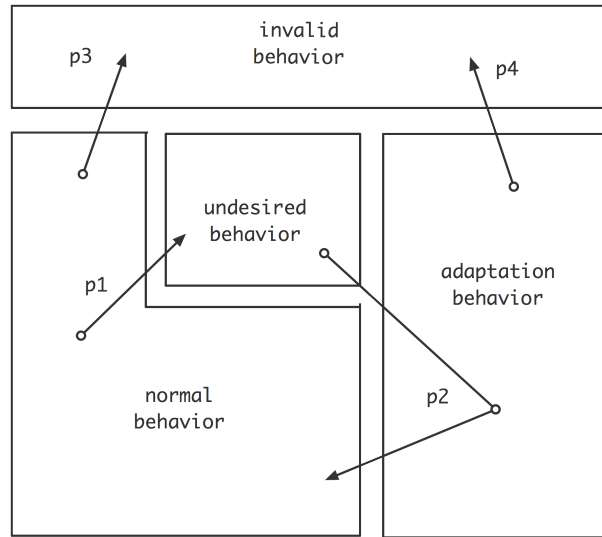

Figure 9: Zones in the state space that represent different behaviors of a self-adaptive system

In the zone *normal behavior*, the system is performing its domain functionality. In our case study this corresponds to monitoring traffic jams. In the zone *undesired behavior*, the system is in a state where adaptation is required. In the case study, this corresponds both to a state where a reorganization is required, or a state where a camera failed. In the zone *adaptive behavior*, the system is adapting itself to deal with the undesired behavior. In the case study, this means either organizations are merging or splitting, or the system heals itself from a failure. Finally, the zone *invalid behavior* corresponds to states where the system should never be, e.g., deadlock in the case study. Properties of interest with respect to self-adaptation typically map to transitions between different zones. For example, property p1 in Fig. 9 refers to a transition from normal behavior to undesired behavior (e.g., property R1). Property p2 refers to the required adaptation of the system to deal with the undesired behavior, that is, the system leaves the undesired state, adapts itself and eventually, returns to normal behavior (e.g., R2 to R4). Properties p3 and p4 are examples of transitions to invalid behavior that should never occur (e.g., I1 and I2).

6   Related Work

We discuss related work on formal modeling of self-adaptation in three parts: fault-tolerance and self-repair, verification of various properties, and integrated approaches. We conclude with a brief discussion of the position of the work presented in this paper.

Fault-tolerance and self-repair. [25] introduces an approach to create formal models for the behavior of adaptive programs. The authors combine Petri Nets modelling with LTL for property checking, including correctness of adaptations and robustness properties. [10] presents a case study in formal modeling and verification of a robotic system with self-x properties that deal with failures and changing goals. The system is modeled as transition automata and correctness is checked using LTL and CTL (computational tree logic). [16] outlines an approach for modeling and analyzing fault tolerance and self-adaptive mechanisms in distributed systems. The authors use a modal action logic formalism, augmented with deontic operators, to describe normal and abnormal behavior. [7] models a program as a transition system, and

present an approach that ensures that, once faults occur, the fault-intolerant program is upgraded to its fault-tolerant version at run-time.

Various properties. [3] presents a verification technique for multi-agent systems from the mechatronic domain that exploits locality. The approach is based on graph, and graph transformations, and safety properties of the system are encoded as inductive invariants. [13] PobSAM is a flexible actor-based model that uses policies to control and adapt the system behavior. The authors use actor-based language Rebeca to check correctness and stability of adaptations. [11] presents a formal verification procedure to check for correct component refinements, which preserves properties verified for the abstract protocol definition. A reachability analysis is performed using timed story charts. [2] considers self-adaptive systems as a subclass of reactive systems. CSP (Communicating Sequential Processes) is used for the specification, verification and implementation of self-adaptive systems.

Integrated approaches. [9] uses architectural constraints specified in Alloy as the basis for the specification, design and implementation of self-adaptive architectures for distributed systems. [18] proposes a model-based framework for developing robotic systems, with a focus on performance and failure handling. The systems behaviour is modelled as hybrid automata, and a dedicated language is proposed to specify reconfiguration requirements. The K-Components Framework [6] offers an integrated design approach for decentralized self-adaption in which the system's architecture is represented as a graph. A configuration manager monitors events, plans the adaptations, validates them, rewrites the graph and adapts the underlying system. [1] presents the PLASTIC approach that supports context-aware adaptive services. PLASTIC uses Chameleon, a formal framework for adaptive Java applications.

Position of our work. In this paper, we focus on modeling and verifying combined properties of flexibility and robustness of a real-world system. To that end, we use a well-established formal method, i.e. model checking via the Uppaal tool. Most existing formal approaches for self-adaptive systems assume a central point of control to realize adaptations. We target systems in which control of adaptations is *decentralized*, that is, managing systems detect the need for adaptations and coordinate to realize the required adaptations locally. Most researchers employ formal methods in one stage of the software life cycle; notable exceptions are [9, 25, 18]. The formal approach used in this paper supports architectural design, but fits in an integrated formally founded approach to validate the qualities of self-adaptive systems that aims to exploit formal work products during subsequent stages of the software life cycle.

7   Conclusions and Challenges Ahead

In this paper, we presented a case study on formal modeling and verification of a decentralized self-adaptive system. The Uppaal tool allowed us to model the system and verify the required flexibility and robustness properties. We defined a dedicated environment model both to verify specific adaptation scenarios and manage the state space problem. This work fits in our long term research objective to develop an integrated approach for formal analysis of decentralized self-adaptive systems that combines verification of architectural models with model-based testing of applications to guarantee the required runtime qualities. As the next step in our research, we plan to build upon the work presented in this paper in two ways. First, we plan to study how we can apply verified architecture models to test concrete implementations using model based testing [19]. As model based testing has mainly focused on functional testing so far, a key challenge here is to extend the approach to test quality attributes. Second, we plan to elaborate on the initial zone-based model of self-adaptive systems that defines the different types of behavior of this class of systems. To that end, we are currently performing a systematic literature review on model checking of self-adaptive systems to map existing work on the model.

References

[1] M. Autili, P. Di Benedetto & P. Inverardi (2009): *Context-Aware Adaptive Services: The PLASTIC Approach*. In: *Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science* 5503, Springer, doi:10.1007/978-3-642-00593-0_9.

[2] B. Bartels & M. Kleine (2011): *A CSP-based framework for the specification, verification, and implementation of adaptive systems*. In: *Software Engineering for Adaptive and Self-Managing Systems*, ACM, doi:10.1145/1988008.1988030.

[3] B. Becker, D. Beyer, H. Giese, F. Klein & D. Schilling (2006): *Symbolic invariant verification for systems with dynamic structural adaptation*. In: *28th International Conference on Software Engineering*, doi:10.1145/1134285.1134297.

[4] G. Behrmann et al. (2006): *UPPAAL 4.0*. In: *International Conference on Quantitative Evaluation of Systems*.

[5] B. Cheng et al. (2009): *Software Engineering for Self-Adaptive Systems: A Research Roadmap*. In: *Software Engineering for Self-Adaptive Systems, LNCS 5525*, Springer, doi:10.1007/978-3-642-02161-9_1.

[6] J. Dowling (2004): *Decentralised Coordination of Self-Adaptive Components for Autonomic Distributed Systems*. Ph.D. thesis, University of Dublin, Trinity College.

[7] A. Ebnenasir (2007): *Designing Run-Time Fault-Tolerance Using Dynamic Updates*. In: *Software Engineering for Adaptive and Self-Managing Systems*, IEEE, doi:10.1109/SEAMS.2007.5.

[8] D. Garlan, S. Cheng, A. Huang, B. Schmerl & P. Steenkiste (2004): *Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure*. *IEEE Computer* 37(10), doi:10.1109/MC.2004.175.

[9] I. Georgiadis, J. Magee & J. Kramer (2002): *Self-organising software architectures for distributed systems*. In: *1st Workshop on Self-healing Systems*, ACM, doi:10.1145/582129.582135.

[10] M. Gudemann, F. Ortmeier & W. Reif (2006): *Formal Modeling and Verification of Systems with Self-x Properties*. Lecture Notes in Computer Science, Springer, doi:10.1007/11839569_4.

[11] C. Heinzemann & S.Henkler (2011): *Reusing dynamic communication protocols in self-adaptive embedded component architectures*. In: *14th Symposium on Component-based Software Engineering*, ACM, doi:10.1145/2000229.2000246.

[12] J. Kephart & D. Chess (2003): *The Vision of Autonomic Computing*. *IEEE Computer Magazine* 36(1), doi:10.1109/MC.2003.1160055.

[13] N. Khakpour, R. Khosravi, M. Sirjani & S. Jalili (2010): *Formal analysis of policy-based self-adaptive systems*. In: *Symposium on Applied Computing*, ACM, doi:10.1145/1774088.1774613.

[14] J. Kramer & J. Magee (2007): *Self-Managed Systems: An Architectural Challenge*. Future of Software Engineering, doi:10.1109/FOSE.2007.19.

[15] R. de Lemos et al. (2012): *Software Engineering for Self-Adaptive Systems: A Second Research Roadmap*. In: *Software Engineering for Self-Adaptive Systems, LNCS 7475*, Springer.

[16] J. Magee & T. Maibaum (2006): *Towards specification, modelling and analysis of fault tolerance in self managed systems*. In: *Self-adaptation and Self-managing Systems*, ACM, doi:10.1145/1137677.1137684.

[17] P. Oreizy, N. Medvidovic & R. Taylor (1998): *Architecture-based runtime software evolution*. In: *International Conference on Software Engineering*, doi:10.1109/ICSE.1998.671114.

[18] L. Tan (2006): *Model-Based Self-Adaptive Embedded Programs with Temporal Logic Specifications*. In: *6th International Conference on Quality Software*, doi:10.1109/QSIC.2006.41.

[19] M. Utting & B. Legeard (2007): *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann.

[20] M. Utting et al. (2011): *A taxonomy of model-based testing approaches*. *Testing, Verification and Reliability*, doi:10.1002/stvr.456.

[21] E. Vassev & M. Hinchey (2009): *ASSL: A Software Engineering Approach to Autonomic Computing*. *Computer* 42(6), pp. 90–93, doi:10.1109/MC.2009.174.

[22] D. Weyns (2012): *Towards an Integrated Approach for Validating Qualities of Self-Adaptive Systems*. In: *ISSTA Workshop on Dynamic Analysis (WODA)*, doi:10.1145/2338966.2336803.

[23] D. Weyns, R. Haesevoets, A. Helleboogh, T. Holvoet & W. Joosen (2010): *The MACODO Middleware for Context-Driven Dynamic Agent Organzations*. ACM Transactions on Autonomous and Adaptive Systems 5(1), doi:10.1145/1671948.1671951.

[24] D. Weyns, S. Malek & J. Andersson (2012): *FORMS: Unifying Reference Model for Formal Specification of Distributed Self-Adaptive Systems*. ACM Transactions on Autonomous and Adaptive Systems, *7(1)*, doi:10.1145/2168260.2168268.

[25] J. Zhang & B. Cheng (2006): *Model-based development of dynamically adaptive software*. In: *28th International Conference on Software Engineering*, ACM, doi:10.1145/1134285.1134337.