Engineering, 2015, 7, 625-643
Published Online October 2015 in SciRes. http://www.scirp.org/journal/eng
http://dx.doi.org/10.4236/eng.2015.710056

A Study on Configuration and Integration of Sub-Systems to System-of-Systems with Rule Verification

Tim Warnecke

Department of Computer Science, TU Clausthal, Clausthal-Zellerfeld, Germany
Email: tim.warnecke@tu-clausthal.de

Received 18 June 2015; accepted 23 October 2015; published 26 October 2015

Abstract

Increasing complexity of today's software systems is one of the major challenges software engineers have to face. This is aggravated by the fact that formerly isolated systems have to be interconnected to more complex systems, called System-of-Systems (SoS). Those systems are in charge to provide more functionality to the user than all of their independent sub-systems could do. Reducing the complexity of such systems is one goal of the software engineering paradigm called component-based software engineering (CBSE). CBSE enables the developers to treat individual sub-systems as components which interact via interfaces with a simulated environment. Thus those components can be developed and implemented independently from other components. After the implementation a system integrator is able to interconnect the components to a SoS. Despite this much-used approach it is possible to show that constraints, which are valid in an isolated sub-system, are broken after this system is integrated into a SoS. To emphasize this issue we developed a technique based on interconnected timed automata for modelling sub-systems and System-of-Systems in the model checking tool UPPAAL. The presented modelling technique allows it to verify the correctness of single sub-systems as well as the resulting SoS. Additionally we developed a tool which abstracts the complicated timed automata to an easy to read component based language with the goal to help system integrators building and verifying complex SoS.

Keywords

System-of-Systems, CBSE, Verification, Model Checking, UPPAAL

1. Introduction

Increasingly demanding user requirements for modern software systems have led to the complexity of the systems being developed becoming more unmanageable. This tendency has been rapidly increasing not just with pure software systems but also with a large number of cyber-physical systems. With the advent of IPv6 and the associated "Internet of Things", another layer of complexity has been added. This new level is also referred to and described as System-of-Systems (SoS). This entails separate and independent systems being linked to a larger system to provide the user with enhanced services and information.

There are many approaches in research and industry used to control the ubiquitous problem of the rapidly increasing complexity of software systems. One is the component-oriented software development approach [1], which divides the system into individual software components with clearly defined responsibilities and assigns well-defined interfaces. One advantage of this approach is that the development of individual components can take place independently of one another. This is achieved by ensuring that the components needed for execution are simply simulated with the desired values or entries. The real components are only connected to a joint system later on in the development process. However, the component-oriented approach may also apply to the SoS mentioned previously, since the isolated individual systems may again be regarded as components that provide services for other systems or require these from other systems.

Nonetheless, although the development of complex software systems has become more structured using methods such as component-oriented software development, these methods do not guarantee the correctness of either the components that are to be developed or the way in which they interact in the overall system. Here verification procedures and tools are used that test systems for certain specifications and constraints to prove whether these are valid or not. Such procedures allow to identify (critical) design errors during the implementation or planning phase and to correct these before they are implemented in the system.

One such tool for software verification is the model checker UPPAAL [2] which is able to simulate system models and check whether these models satisfy certain specifications. Due to the type of modelling, based on networked and timed automata, this tool is used principally to verify real-time systems and communication protocols. The modelling based on networked automata means that UPPAAL is also an interesting tool for modelling networked software components and systems.

This work therefore investigates the suitability of UPPAAL in terms of component-oriented development and the linking of components to a System-of-Systems. It will demonstrate how UPPAAL can be used to model individual sub-systems and SoSs. It will also exhibit that specifications that are valid in a sub-system do not necessarily have to be valid in the interconnected SoS. In order to establish this, the study will first model the sub-systems in UPPAAL and then link the sub-systems to a functioning SoS using a self-developed tool.

2. Component-Based Development of System-of-Systems

The difficulty of verifying the correctness of a software system in terms of a given specification should not be underestimated. The use of the component-oriented development approach offers the possibility of reducing the complexity of both the integral system that is to be developed and its verification. This is achieved by dividing the system into smaller and more manageable components. Verification can then be carried out on these smaller system components. However, it would be misleading to conclude that this process would ensure the correctness of the entire system.

First we introduce the chosen type of component-oriented development for System-of-Systems with a closer look to the left side of **Figure 1**. Initially a component (grey-coloured squares) is only a black box and its internal behaviour cannot be observed from the outside. The only thing that is known about a component are the well-defined interfaces which provide either services and/or information (green rectangles). Other components can use or require these (red rectangles), so that they are able either to perform their task or perform it better than they would otherwise be able to. If a component defined as such were to be developed separately from the other parts of the system, the values for the required interfaces will have to be provided by a dummy component.

This dummy component is also referred to as the expected environment (blue-colored squares), as it simulates the environment which is expected by the component in order to be able to function correctly. If the interfaces of the component and its expected environment are connected using communication channels, a dedicated sub-system is created, since this system can now fully perform its functionality independently of other systems. Based on this assumption, such a sub-system can also be tested to verify the correctness of certain properties using verification procedures, e.g. model checking. Accordingly, as shown in definition 1, a sub-system is understood to embrace a combination of the components to be developed, the expected environment, the channels

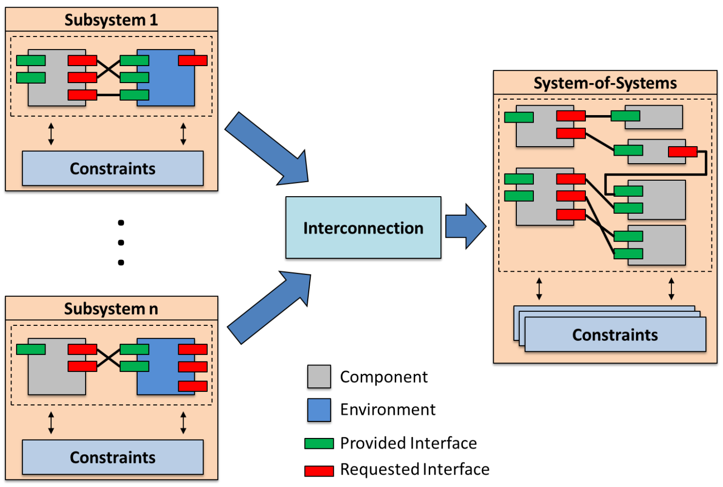

Figure 1. Integration of sub-systems into a System-of-Systems.

between the interfaces of these two and any number of constraints.

Definition 1: A **sub-system *T*** is the 4-tuple (*Comp*, *Env*, *Chans*, *Cons*) where

- ***Comp*** is the component to be developed with any number of interfaces that are required (req_{Comp}) and provided ($prov_{Comp}$).
- ***Env*** is also a component with interfaces that are required (req_{Env}) and provided ($prov_{Env}$) and is called the environment. The environment covers all interfaces which are expected by *Comp*. It offers the component these interfaces.
- ***Chans*** is a set of channels connecting the interfaces of *Comp* and *Env*, where $req_{Comp} \times prov_{Env}$, but not necessarily $prov_{Comp} \times req_{Env}$.
- ***Cons*** is the set of constraints φ, which have to apply for connecting *Comp* and *Env*.

The System-of-Systems to be designed (see **Figure 1**, on the right) logically comprises the combined set of sub-systems where the expected environments are omitted. The previous interconnection between the components with their environment (represented by the set of channels) is of course no longer valid using this process. How the validity of the channels can be restored is one of the questions which arise when an integral system has to be designed from sub-systems. Definition 2 provides a more formal specification of the meaning of a SoS.

Definition 2: A **System-of-Systems *S*** is the 3-tuple (*Comp*, *Chans*, *Cons*) where

- ***Comp*** is the combined total of all components, *i.e.* $\bigcup_{i=1}^{n} Comp_i$, from the selected sub-systems T_i.
- ***Chans*** is the set of channels connecting the interfaces of the integrated components, where $req_{T_i} \times prov_{T_j}$ and *i*, *j* are pairwise different.
- ***Cons*** is the combined set of constraints, *i.e.* $\bigcup_{i=1}^{n} Cons_i$, from the selected sub-systems T_i.

The constraints of the sub-systems constitute an additional problem when it comes to integrating them into a SoS. This is because they often relate to the previous environments. However, they no longer exist and therefore no longer function on a syntactic and semantic level. On the other hand, it may happen that the constraints can no longer be maintained in the SoS due to details not taken into account in the design of the environment. Nevertheless, the constraints of the individual sub-systems must also be included in the SoS and be integrated accordingly. This study does not address this factor but demonstrates that even small systems and simple constraints can lead to major complications.

3. Introduction to Formal Verification of Software Systems

The formal verification of software systems is an area of research that has been going on since the 1980s. It has enjoyed increasing popularity and use even in industry due to numerous disasters and mishaps caused by software and hardware errors. In this area, a distinction can basically be made between verification at runtime and design time. While runtime verification monitors a system during execution and therefore requires a fully implemented program, the verification at the design time can already be applied during the design phase. This study focus on the latter one and especially on a model checking tool named UPPAAL. The task of such a verification tool used at the design time is automated system checking against an often safety-critical specification. Both model checking and UPPAAL are explained in detail in the following sections.

3.1. Model Checking

One of these formalized verification approaches is model checking, which was introduced first in [3]. The idea of model checking is to find states in a Kripke structure [4] that apply to a given specification. A Kripke structure is a variant of a non-deterministic automaton and is defined in (1) as follows as a 5-tuple

$$K = \left(S, S_0, R, L, AP\right) \tag{1}$$

where S is a finite set of states, $S_0 \subseteq S$ is a finite set of initial states, $R \subseteq S \times S$ is a left-total transition relation, $L : S \rightarrow 2^{AP}$ is a labeling function, and AP is a set of atomic propositions.

The established structure which is equivalent to an abstract machine can help to model the processes required in a program. The interpretation of these processes is generally performed using formulas, which are expressed in temporal logic [5]. Temporal logic is a form of modal logic [6], which already allows conclusions in statements and describes temporal relationships between events. Based on temporal logic, two different categories of description techniques have emerged over time known as Linear Temporal Logic (LTL) and Computational Tree Logic (CTL). Although both types of logic make statements about whether a sought-after event will occur in the future, LTL is based on the concept of paths and CTL used branches or trees as its basis. CTL, which in turn is a subset of the more general CTL* [7], will be discussed in more detail below. An extension of CTL, Timed Computation Tree Logic (TCTL), is used in the UPPAAL model checker, which will be introduced later.

As already explained, in CTL all possible execution paths of a program are represented by a tree in which the nodes represent the predefined states of the Kripke structure K and the edges represent state transitions of the transition relation R. The root corresponds to an initial state $s_0 \subseteq S_0$. The leaves, however, represent deadlocks. A state is known as a deadlock when no further transition can take place, therefore terminating the execution of the program. However, this is not wanted in many programs and therefore a frequently tested characteristic.

CTL defines a number of operators with which specifications can be described. These operators can initially be divided into path and temporal quantifiers. The two path quantifiers All A and Exists E denote the number of paths to which a temporal logical formula φ must apply. The term $A\varphi$ therefore states that formula φ must apply to all paths of a tree and $E\varphi$ indicates that there must be at least one path to which θ applies. However, this type of path quantifier can never be used without a temporal quantifier in CTL, as it is only possible in the more general CTL*. These four temporal quantifiers are called Next (X), Finally (F), Globally (G) und Until (U) and indicate when a given formula φ should apply. The unary temporal quantifiers Next, Finally und Globally indicate that formula φ must apply in the next state (X), in any next state (F) or in all following states (G). The single binary quantifiers Until states that a state s_1 applies until a new state s_2 is reached.

According to the definition (1) and the temporal logics operating on them, the problem to be solved by model checking can be formally defined accordingly [8] as formula (2)

$$K, S' \vDash f \tag{2}$$

where K is a Kripke structure, $S' \subseteq K$ is a set of K-based states, and f is a temporal logical formula.

This means that based on the two parameters K and f, all S' states of the K Kripke structure will be sought, in other words the expression $K, S' \vDash f$ has to be true. In order to identify these valid states S', the Kripke structure (which comprises all possible program processes) will first be generated, and then uninformed search methods will be used on these. However, this is not trivial in practice as the size of a Kripke structure relates exponentially to the description of the modelled system [9]. This problem is known as state space explosion and ensures that the Kripke structure will either no longer be completely written into the working memory or that the search for desired states can take a very long time. Many scientific works aim to mitigate this problem and cover, for example, the use of symbolic model checking [10], partial order reduction [11], CEGAR (Counterexample-guided abstraction refinement) [12] and some other methods which all have the common goal of keeping the state space as small as possible.

3.2. Model Checking with UPPAAL

As already mentioned model checking has become very popular and so a lot of tools were developed. One of them is UPPAAL which the developers label as an “integrated environment for modelling, validating and verifying real-time systems” [13]. For modelling, the developers focused on networked and timed automata that have been enriched by data types. These can be created and manipulated with a simple graphical interface. Validation and simulation of the execution of the modelled systems take place in an environment similar to modelling. This means that the modeller can view the graphical automata at runtime and thereby see which edges to switch in the next execution step or which states will be accepted by the automata. The last area, verification, is the actual model checking. At this point, requirements may be imposed on the modelled system and checked by UPPAAL.

The UPPAAL model checker refers to automata as templates, as this highlights that automata that have been modelled once can be used repeatedly. The reuse of such automata not only shortens modelling time, but also significantly simplifies their maintenance. The modelling of the templates is also complemented by two textual descriptions that differ in terms of scope and visibility. The first description is the global declaration, which contains variables and functions, which are visible to all templates in an UPPAAL program. Each template also has its own local declaration which only includes visible variables and functions for the template in question.

The user has two objects available for the modelling of templates: *locations* and *edges*. Locations are the possible states of a program and as shown in **Figure 2**, they are depicted by a simple circle. Such a location may have an optional name and an optional invariant. While the name of a location only facilitates better understanding and clarity, the invariants describe the latest point at which a location has to be exited again. This is exactly the case when an invariant is violated.

A location can take on four different characteristics: *initial*, *urgent*, *committed* or *normal*. An initial location designates the entry point to a template when the program starts. An urgent location refers to the timed approach of automata used in UPPAAL. If the developer has modelled clocks, an urgent location can be used to indicate that no time passes while the system is in this state.

The committed location is an escalation of the urgent location and therefore has the same characteristic with only one addition: A committed location does not just stop time, but it also signals that the location has to be exited again in the next execution step. The last characteristic that a location may have is normal. This only means that it has none of the three aforementioned characteristics.

The entry and exit of locations in automata take place via edges. Like the locations, they can also have certain characteristics in the form of annotations, known as *select*, *guard*, *sync* and *update*. The select-instruction is able to assign a variable with a random value. This allows to simulate non-deterministic user inputs and thereby trigger behaviour which perhaps may not have been taken into account at the present time of development. To prevent an edge from being enabled at any time by the system, the guard instructions will be used. These correspond to conditions that have to be met so that an edge can be considered as a candidate for a state transition. Such a condition is constructed using comparison operators and logic operations.

Communication between two or more automata is one of the fundamental concepts in UPPAAL and means that individual templates are able to synchronize. The synchronisation instruction (*sync* for short) has therefore been introduced to allow edges to model both as a synchronisation senders and synchronisation receivers. Such an instruction consists of the name of the channel via which the synchronisation is to take place and appended to this is either an exclamation mark for a sending edge or a question mark for a receiving edge. However, the desired channel name must first be made known as a variable in the global declaration.

Manipulation of variables is achieved using the update instruction in two different ways. The first possibility is direct manipulation of a variable value with help of mathematical operations. In order to make such manipulations reusable there are also functions in UPPAAL, which can be accessed from the update command. These functions are defined in the global or local declaration and can therefore be accessed like functions, provided that they are within the visibility range of the template concerned.

4. Introduction to the Example System

Based on the theoretical review of the approach to the integration of individual, dedicated sub-systems into a System-of-Systems and the resulting problems in the channels and the constraints, this study will now illustrate these based on a simple example. The example introduces the case of a simple SoS called personnel management system comprising three components which are outlined in **Figure 3**. As already discussed, the components have both interfaces that provide services (prefix r_) and those that require services (prefix p_). The

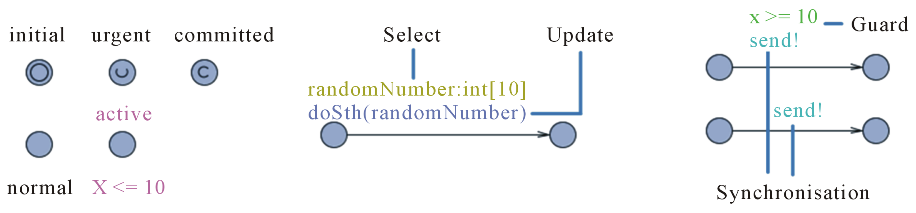

Figure 2. UPPAAL: Graphical representation of various locations and edges.

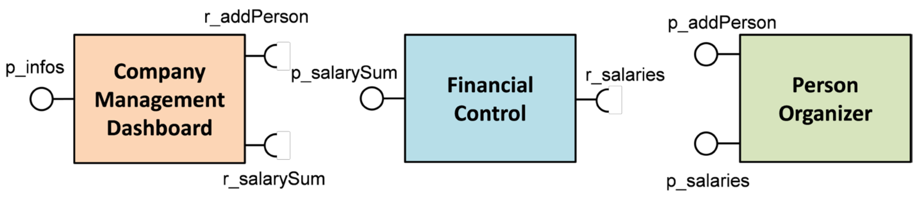

Figure 3. Components of the personnel management system.

syntactical distinction is, however, not just meant for the reader. It should also denote that the interfaces of the components must not be given the same name, if they are going to be connected.

The first component named Company Management Dashboard (CMDB) is the graphical interface of the personnel management system. The user can use this to enter new staff with their salaries and can request the sum total of salaries for all the staff. For this purpose, the component also has three interfaces which perform the actions described. The interface p_infos provides the graphical interface. As the names already suggest, interface r_addPerson requires a component for the comprehensive storage of the personal data and interface r_salarySum needs the result of the sum calculation of all salaries of the persons entered.

The component Financial Control is responsible for the accounting system of the SoS. It has the ability of calculating the salaries of all staff based on a given payroll. This calculation is performed within the component and is made available to other parts of the system at the interface p_salarySum. The lists containing the salaries of the staff needed for the calculation algorithm is requested via interface r_salaries by another component.

The final component is the Person Organizer which represents an electronic data management system, such as a database. In this example, this component is a pure service provider. This means that it only offers services, and requires none. The first of the two interfaces named p_addPerson offers other components the possibility of storing personal data, *i.e.* the person's name and their salary. With interface p_salaries, a component can retrieve all the currently stored personal data as a list.

5. Modelling Components and Expected Environments

Planning and developing a System-of-Systems is a major undertaking and can only be carried out with difficulty at a single central location. This is due to the fact that the systems involved often already have very complex structures and because the domains in which they are located are very different. Furthermore, existing sub-systems in a SoS are mixed with newly developed sub-systems.

It is therefore a good idea to allocate the design and implementation of new sub-systems to different developer teams because this helps to reduce the complexity of the development process. The dummy components mentioned in chapter 2 are required so that these can be developed separately from other sub-systems. These expected environments simulate the parts of the SoS, which provide information and services for their own sub-system. In order to prove that the sub-systems are correct, formal mechanisms like model checking and tools like UPPAAL are used. However, before UPPAAL can be used for such a task, the very informal sounding description of the sub-systems and their interfaces needs to be converted into UPPAAL's modelling language. This study has therefore established some simple but general rules according to which the sub-system descriptions can be translated into UPPAAL automata.

According to convention, only one sub-system per UPPAAL program, containing component and environment, will be modelled. This not only preserves the clarity, but also helps to keep the state space as small as possible during verification. In a distributed system, this is also far more realistic than would be the case if all the sub-systems were available. As already stated in section 3.2, this type of UPPAAL program first contains a global declaration and any number of templates combined with the same number of local declarations. Another rule stipulates that the component and its environment should be modelled as stand-alone automata. This means that there are always two templates in a program. The local declarations associated with the templates carry out the functions which are called at the interfaces of the component or environment. They also carry out other functions which may be needed for the system flow in the components. The nature of UPPAAL is such that when communication takes place between two or more templates, only synchronisation instructions (*i.e.* no values) are exchanged. This means that global variations have to be used for exchanging data. Accordingly, all values that are exchanged between a component and its environment will be processed by variables which are defined in the global declaration.

Modelling a single software component in UPPAAL begins with the creation of an initial location into which the automaton will keep returning after one of its interfaces has been called. The further setup of the automata depends on how the component itself is described by its interfaces. The simplest is the presentation of an interface providing a service. Here an edge is created starting from the initial location and returning to it (see **Figure 4** on the left). The functionality offered by this edge is entered as a function in the edge's update property and implemented in the local declaration of the template.

The situation is different if an interface has to be modelled which calls another component, because it is then necessary to distinguish between *synchronous* and *asynchronous calls*. If another component is called asynchronously, it will also be modelled as an edge which again returns to it directly from the initial state. It is also again defined as a function via the update property. As shown in **Figure 4** (middle), the only difference is that the starting point of a synchronisation channel is added to this edge so it can start calls to other components. This call then returns directly and the component is ready for further actions. If parameters are also transferred in this call, this can be randomized by a select instruction in order, for example, to simulate user input.

In contrast to an asynchronous call, a synchronous call is modelled in a different way and in this case used for a different purpose. In this study, the synchronous calls are therefore used to obtain variable values from other components. This is necessary because when a synchronisation between templates takes place, UPPAAL has a fixed order of function executions. If an edge that has a synchronisation starting point is called, first the update instruction and then the synchronisation instruction of the sending template will be called. Subsequently, in the received template, first the synchronisation instruction and then the update instruction will be processed. This means that the received template is not able to calculate the requested variable value and bind this simultaneously to the global communication variable. In this case, synchronous calls will therefore be used which initially call and then block synchronisation in order to wait for the result. In order to model this call, two edges and a committed location will be used, as shown in **Figure 4** (right). The first edge points from the initial location to the committed location and marks the starting point of a synchronisation channel. This call signals to the called component to calculate the desired values and to bind to the global communication variable. The calling automaton is then located in the committed location, which it exits directly in the next execution step via the second edge in the direction of the initial location. In this way, the actual function is called and the value of the global variable is copied into a local variable of the component.

The environment component is still needed to enable the component to call services. Presenting an environment using an automaton is quite easy to do, as at this point no account has to be taken of the different forms of call. Each interface is going to be provided by the environment and will be presented as synchronous call. Furthermore, the edge with the end of the required synchronisation channel is annotated with a channel-statement in order to enable communication with the component. The interfaces of the environment are indeed necessary if the component requires this. However, it may also be the case that a component is only a service provider. In

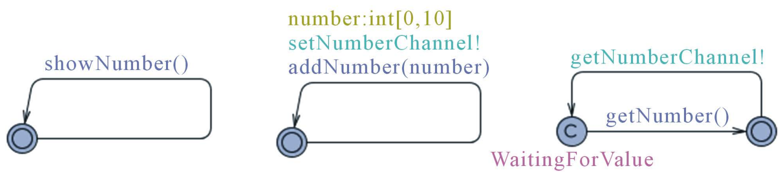

Figure 4. A component's service providing interface (l.), asynchronous call (m.) and synchronous call (r.).

this case, it requires no further services or data and can therefore be modelled fully without an expected environment.

6. Verification of Single Software Components and Their Expected Environments

The division of sub-systems into component and environment is used in the following for the individual components in the example from chapter 4 and then implemented using UPPAAL's automaton language. The component which is modelled exemplary in the manner described below is the Company Management Dashboard (the other two components are modelled accordingly). As this component has two interfaces which require services from other sub-systems, the environment must have two interfaces which provide the desired functionalities (**Figure 5** left). Interface r_addPerson is therefore able to store the personal data, and r_salarySum contains the total of the salaries in order to be able to display this with the graphical interface p_infos. Consequently, the environment is only the combination of the two non-usable components Financial Control and Person Organizer and their provided interfaces.

Based on this informal description of the sub-system, a transfer into the UPPAAL automaton language takes place. The component and its environment as respective templates are illustrated in **Figure 5** (right), where the component is placed at the top and the environment at the bottom. The component's automaton contains three parts representing the interfaces. The first p_infos has been modelled with a simple edge as this can be offered to other sub-systems, in this case the user. The function r_addPerson uses a synchronous call to transfer the parameter ID and salary to the environment. The parameters are selected non-deterministically and synchronised using the channel addPersonChannel. The third interface r_salarySum is used to obtain the sum of the. The component uses this to synchronise with the environment with channel getSalarySumChannel and then obtains the value.

As already indicated, the modelling of sub-systems as UPPAAL automata has the crucial advantage of ensuring that properties can be verified using constraints or specifications. This ability is used to show that it makes a significant difference whether such a constraint is applied to a single sub-system or a jointed SoS. An example constraint which checks an invariant property of the CMDB component is therefore defined in the following. This invariant should indicate that the salary sum, which provides the environment as a value, is always the same as it would be if CMDB were to calculate this itself. In UPPAAL's specification language, such a requirement could be formulated as follows:

$$A[]\, Cmdb.calculateSalarySum\,(CmdbEnv.salaryList) == CmdbEnv.p\_salarySum() \qquad (3)$$

Equation (3) describes how the invariance of the constraint is achieved by the expression $A[]$ which is equal to the statement AG in TCTL. As already explained, this means that a formula φ must apply to all execution paths and in all states. Subsequently, the CMDB component calls the calculateSalarySum function, which takes the environment's salary list (salaryList) as a parameter (here referred to as CmdbEnv) and calculates the total from

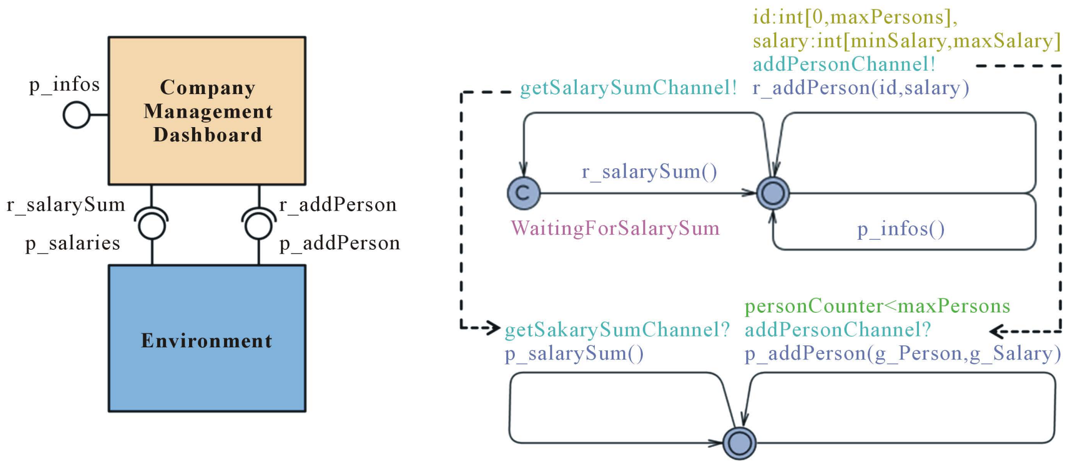

Figure 5. The Company Management Dashboard sub-system as an UPPAL automaton.

this. It should be noted that this function is only used for this constraint and within the component and is therefore not included in the associated template. The calculated salary sum is compared via the equality operator with the sum provided by the environment's p_salarySum interface. If the UPPAAL verifier is executed with the stipulated constraint, it will report that this is always valid. The sub-system has therefore been designed and constructed correctly from the developer's perspective.

7. System-of-Systems Integration

Having addressed the question of modelling single sub-systems, the next question is how they can be integrated into a System-of-Systems which contains of three parts: components, channels and constraints. The first step is to collect all modelled sub-systems and to remove all dummy components respectively expected environments. The result is shown in **Figure 6**.

However, this procedure, only ultimately results in individual components no longer being able to function because the interface channels are no longer correctly connected. The channel ends must therefore be adjusted unilaterally, as the starting points are already registered correctly at the edges. Of course, knowledge of the structure of the automata and their importance as networked components is indispensable. This is because the received synchronisation instructions have to be annotated to the correct edges that are currently not equipped with this instruction in order to produce the interconnection between the components. Unwanted select instructions must also be removed. Otherwise, the SoS would have inconsistent data.

A possible outcome of this manual adjustment of automata is shown in **Figure 7** and has been achieved adding three new channel ends to the edges p_salarySum, p_salaries and p_addPerson. The correct designation of the ends has meant that these have been connected to their respective counterparts. The select statement of the Person Organizer has also been removed, as the random inputs have now been reprocessed via the CMDB.

After the components have been interconnected, the set of constraints which were initially and knowingly not taken into account, remain. Without loss of generality, in this example the set contains only a single constraint. This has already been mentioned in the previous section and verified a characteristic of the CMDB component. The constraint in Equation (3) was formulated as follows:

$$A[]\, Cmdb.calculateSalarySum\,(CmdbEnv.salaryList) == CmdbEnv.p\_salarySum()$$

The question that now arises is whether the constraint, which is correct for the CMDB component, can still be valid in the constructed SoS. First, it can be stated that the constraint is no longer verifiable, as the template CmdbEnv (the environment of CMDB) is no longer available. The two references to CmdbEnv must therefore be replaced by components used in the new interconnection of the SoS. In the first verification, CmdbEnv represented the data from the Person Organizer (PO), as CMDB is meant to calculate the sum total with the salaries

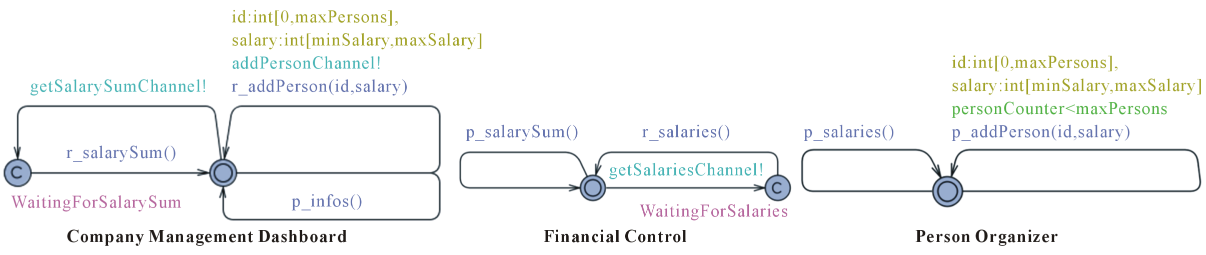

Figure 6. The personnel management system as non-interconnected UPPAAL automata.

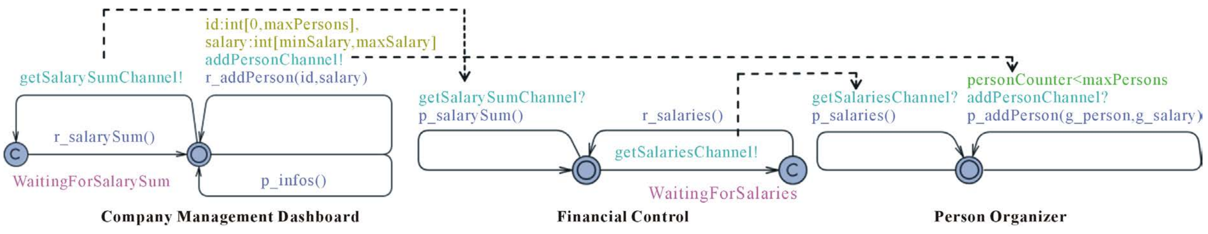

Figure 7. The personnel management system as interconnected UPPAAL automata.

that are currently available using the calculateSalarySum method. The CmdbEnv behind the equality operator represents the component which will take over the calculation of the salary sum. In this case it is Financial Control (FC) and given the assumptions and observations that have been made, the constraint appears as follows:

$$A[]\,Cmdb.calculateSalarySum\,(PO.salaryList) == FC.p\_salarySum() \tag{4}$$

Equation (4) describes the constraint adjusted to the SoS can now be verified again by UPPAAL with the new result that this is no longer valid. The reason for this lies in the interconnection of the SoS. In the initial verification of Equation (3), the storage of the personal data and the calculation of the salary sum have been carried out by the same components but this is no longer the casein Equation (4). These responsibilities have been separated and therefore so has the information. If the user now enters personal data into the system, this data will first be stored in the PO. At this moment, the constraint is already invalid, because FC does not know anything about the new entry as its interface r_salaries has not yet queried the new salary sum. The amount calculated by FC for one execution step is therefore different to that which CMDB would calculate itself. There are therefore contradictory variable values in the system.

This simple example clearly shows how easily a constraint suddenly loses its validity if applied in a SoS. The situation described only appears very briefly in the personnel management system and only when a user enters new data. However, these brief moments in safety-critical systems might determine whether a fatal error occurs or not. As has happened in the example shown, conflicting variable values, may lead to disasters, as other sub-systems may rely on these values. It is therefore important to have a better understanding of the constraints used and their importance for a SoS, and to develop methods to recognise and resolve these defects in advance.

8. Tool Support

To support the development and verification of component-based software systems a small tool called InterConnection Tool (ICT), shown in **Figure 8**, was developed. The main idea of the ICT is that modelling of components and systems is completely different of modelling timed automata. Because of that the tool transforms subsystems modelled as timed automata to a component-based representation consisting of components, interfaces and channels. Using this representation it is very easy for the system integrator to build a SoS even if he has no knowledge of modelling automata. After building a SoS the ICT transforms it back to automata representation so the system integrator is able to verify the correctness of his design.

The interaction with the ICT is kept as simple as possible, but requires some preparation from the component developer, because first he has to model the sub-systems behavior as timed automata. After he has completed the modelling, simulation and verification of the sub-systems, he stores each component in a separate XML file. This format is already provided by UPPAAL. Subsequently, these files are loaded by the system integrator using the ICT. Once the loading process is complete, all detected components are displayed on the left in the list of inactive components (**Figure 8** on the left).

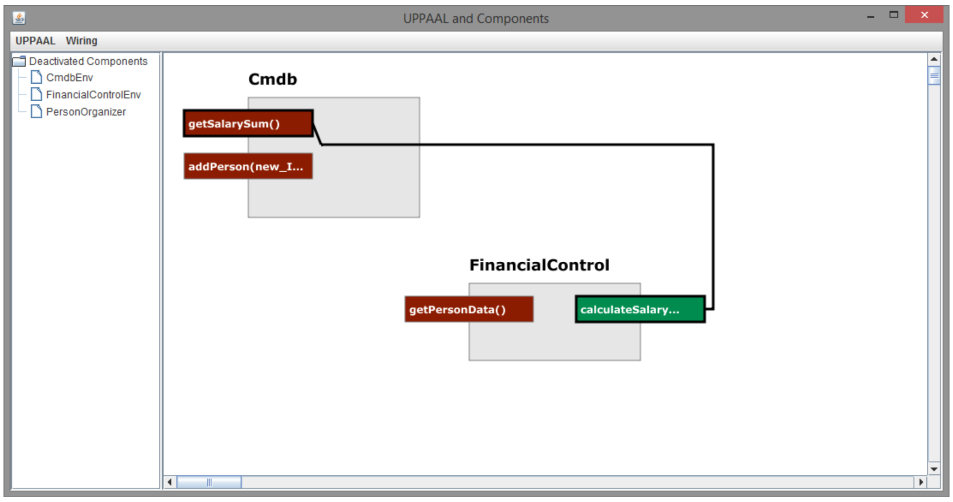

Figure 8. Graphical interface of the Inter Connection Tool (ICT).

The system integrator is now able to drag the components from the list onto the workspace and they will be displayed as grey squares with attached red (required interfaces) respectively green rectangles (provided interfaces). If he selects an interface, a connection edge appears which is anchored at the selected interface and follows the cursor. To complete the interconnection, the interface of another component, that does not match the color of the interface already chosen, must be selected. The user can repeat this process until the interconnection is complete. Afterwards the ICT transforms the result into a new combined UPPAAL automaton. This file can now be loaded in UPPAAL in order to check whether the system still has the required properties.

9. Case Study

In order to test our approach and the ICT presented in the previous chapter in terms of its suitability for common daily use, this study has examined and modelled an example which is more complex than the personnel management system. This chapter therefore outlines a scenario in the area of alerting and dispatching of operational services in emergency situations and then models these as UPPAAL automata based on the rules already described in chapter 5. Three selected sub-systems are then checked for their accuracy using UPPAAL specifications. Once the test is completed, the automata are transformed with the implemented connection tool into the linking language. The components are subsequently interconnected semi-automatically and retransformed back into a single UPPAAL automaton, the System-of-Systems. It is then shown that the SoS is no longer able to meet the combined constraints of the sub-systems.

9.1. Scenario

Any kind of incident that threatens the lives of people and animals is the responsibility of organisations such as the fire, ambulance and police services. These services are responsible for dealing with emergencies that fall under their area of responsibility and taking countermeasures. For example, police officers can arrest the perpetrator of a robbery and the fire service can rescue people involved in a car accident. The coordination of resources within this type of organisation is already a difficult undertaking. It is, for example, possible that there are not enough fire engines at a fire station to be able to deal with a major fire and therefore other fire engines from other fire stations further away will be needed. In this type of case, the coordinators responsible have to react quickly and be able to make a good assessment of the situation.

This problem will be more complex if there are emergencies that have to be dealt with and which involve two or more of the organisations mentioned at the same time. A fire would, for example, require resources from the fire, police and ambulance services, because in this type of operation people are rescued by the fire service and then passed on to the ambulance service. The police must also block off the streets to ensure the smooth running of the rescue operation.

Coordinating these activities that involve the input of various organisations provides a basis for the following case study. As shown in **Figure 9**, this scenario involves twelve parties and four different types of operation, which are described in more detail below.

9.2. Parties Involved in the Scenario

The caller is any person who sets the entire process into motion by calling the control centre. He or she may also be questioned by the control centre regarding the details of the emergency. The control centre receives the call and asks the caller for details regarding the type of emergency and the number of casualties. Once the type of operation has been identified (the types of operation are explained in the following sub-section), the control centre can ask for available resources from the organisations that are required in order to determine whether their services can be called upon. For example, if a fire service has no more fire engines, it is not helpful to call upon these engines to attend to a fire. If there are sufficient resources available, the organisation will be alerted. If this is not the case, the control centre can still access and alert external services. Finally, the control centre can store all reported operational data in a database.

The archive is a software database, which is able to store the operational data reported by the control centre. However, to simplify the example, the operational data only contains the reported operational type.

The external forces function as surrogate support for each organisation that can provide no further vehicles for new operations. In a realistic context, the The Federal Agency for Technical Relief would, for example, provide

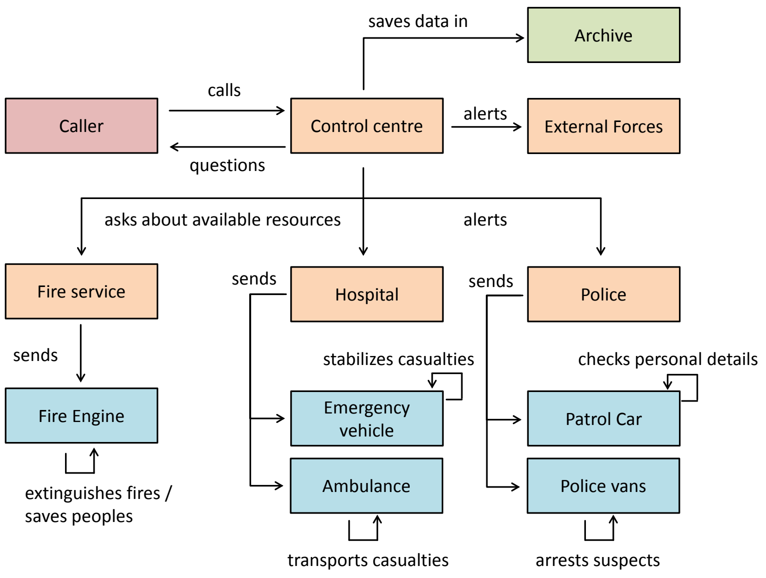

Figure 9. Overview of the case study.

support to the fire service, the Red Cross would provide a back-up to the ambulance service and the armed forces would preserve public order in the last resort if the police were no longer able to do this. In the scenario described here, external forces may, however, only be alerted as the management of the resources for these operational services is not the responsibility of the modelled control centre.

The headquarters for the three organisations (fire service, hospital and police) have a limited contingent of certain types of vehicle that are available to them for dealing with emergencies. The control centre can ask for this contingent at any time. Each particular type of operation costs operational headquarters a certain number of particular types of vehicle, and when no more of these are available, the service concerned will turn down any further operations. These types of emergency vehicles differ depending on the organisation. The fire service may send fire engines for fire fighting. The hospital may provide doctors and ambulances for medical emergencies and the police may provide patrol cars and police vans for combating crime.

The resources deployed in this scenario are the vehicles of each organisation. All vehicles may be sent from their respective headquarters. As already mentioned, every time a vehicle is deployed it will cost the organisation concerned a certain amount of resources. The fire engine is the only vehicle at the fire station and is able to rescue people and extinguish fires. The police maintain two different types of vehicles: the patrol car and the police van. The crew of the patrol car can check a person's identity and the police van is used if there are suspects to be arrested. The last two vehicles are the emergency car and the ambulance belonging to the ambulance service. The emergency car is used for first aid at the scene of the accident. Once first aid has been administered, the patient is transferred to the crew of the ambulance to be taken to the hospital.

9.3. Types of Operations in the Scenario

The case study modelled also differentiates between four types of operation, which are not shown in **Figure 9**. These types of operation relate to false alarms, bank robberies, house fires and medical emergencies. At the beginning of the scenario, the caller selects a random type of operation and communicates this to the control centre. The type of operation chosen determines not only the organisations that are to be alerted but also the number of vehicles that these organisations will send to deal with the emergency.

If the control centre determines that the reported emergency is a false alarm, the control centre will alert none of the organisations and only save the call in the archive. The control centre will not carry out any further steps and the operation will be considered complete.

If the caller reports a bank robbery, the control centre will only alert police headquarters and relay the accompanying operational code. The police station will use the code to send three patrol cars and a police van to deal with the situation.

If the eye witness reports a medical emergency, such as a heart attack, the control centre will only alert the hospital with the associated type of operation. In medical emergencies, the protocol of the hospital is always to send an ambulance in the first instance to administer first aid. An emergency car will follow shortly thereafter to provide the patient with medical care. The patent will then be taken to hospital by ambulance.

House fires are the last type of emergency that can be communicated to the control centre. This emergency involves all three organisations being alerted, as people have to be rescued and treated, the fire has to be extinguished and the traffic has to be redirected. Once the alarm has been raised, two fire engines, two emergency cars, two ambulances and two patrol cars are deployed from the operational centres.

9.4. Modelling of the Sub-Systems

After introducing the scenario, the specified expertise of the parties involved and the types of operation are transferred using the rules described in chapter 5 to UPPAAL automata and their expected environments are modelled. This also applies to systems which not only consist of software and/or hardware. This means that even people such as callers are regarded as sub-systems in order for them to be integrated into the overall System-of-Systems. The order of the following UPPAAL automata corresponds to the order of the parties mentioned in the previous section. It should be mentioned again at this point that sub-systems that only provide services, but do not require any more themselves, manage without modelled environments.

The caller is shown with his scope for action as an UPPAAL automaton in **Figure 10**. The caller is able to make an emergency call with the edge callEmergency and offers to communicate information regarding the type of emergency via provideData (emergencyType). Accordingly, this automaton only expects the environment to accept his emergency call and later ask about the type of emergency service required.

The control centre is not only the centre of the entire System-of-Systems, but also provides and requires most of the services in this scenario. Accordingly, the UPPAAL automaton, shown in **Figure 11**, is much more complex. The control centre can use the edges emergencyCall and getEmergencyData to receive a person's call and find out about the type of operation that is needed. The control centre can then use the edges with the suffix Resources, such as getFireHQResources or getPoliceResources, to enquire about available resources for the organi-

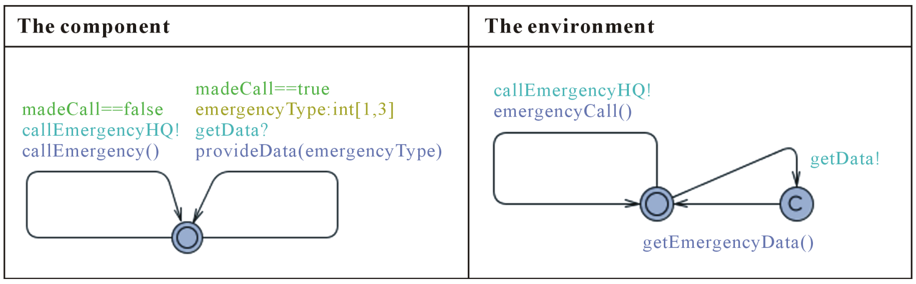

Figure 10. The caller and the corresponding environment as an UPPAAL automaton.

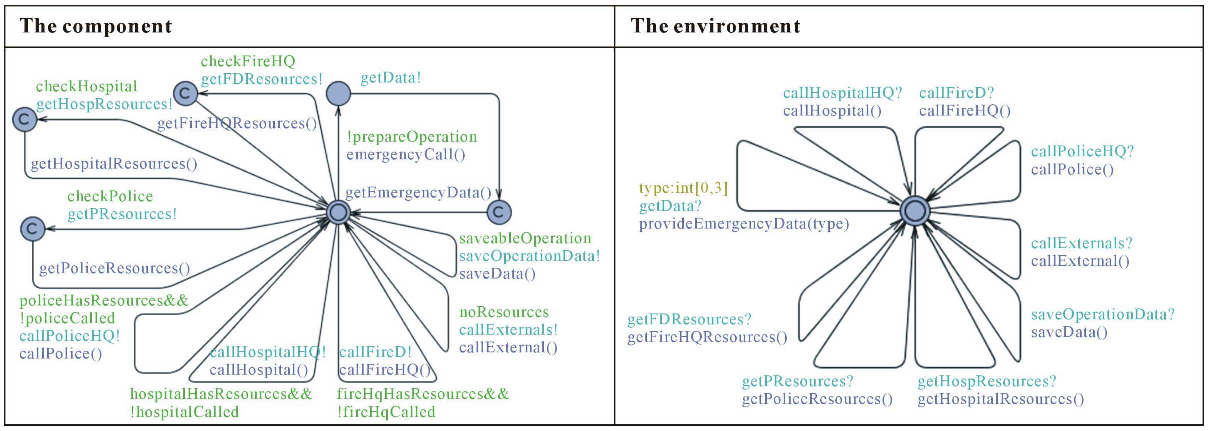

Figure 11. The control centre and the corresponding environment as an UPPAAL automaton.

sation concerned in order to alert the organisation (edges with the prefix call). However, this is only possible if headquarters still has vehicles available. If this is no longer the case, external forces will be called via the edge callExternals. The control centre can also save operational data at any time via the edge saveData in the archive.

The modelled environment of the control centre is therefore able to provide the number of vehicles for the three organisations and accept alerts from the control centre. Furthermore, the environment transmits incidental types of operation to the control centre and it serves as a database for the operational data.

In this scenario, both the archive and the external forces are very limited in terms of their functionality, as they can either only save data or be alerted. As shown in **Figure 12**, this leads to very simple automata that have no expected environment because they do not require services from other components.

The descriptions of the fire service, the hospital and the police suggest that the functionality of these three organisations is very similar. Accordingly, the UPPAAL automata in **Figure 13** also resemble one another very

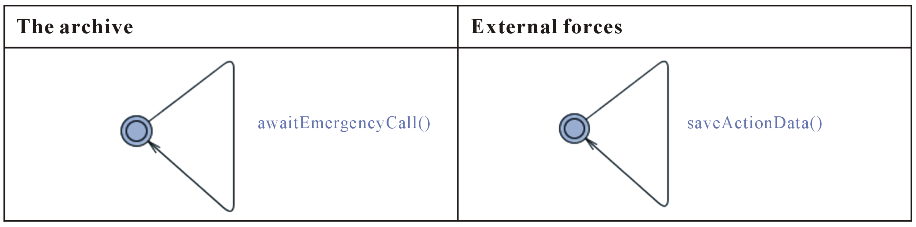

Figure 12. The archive and external forces as UPPAAL automata (no environment needed).

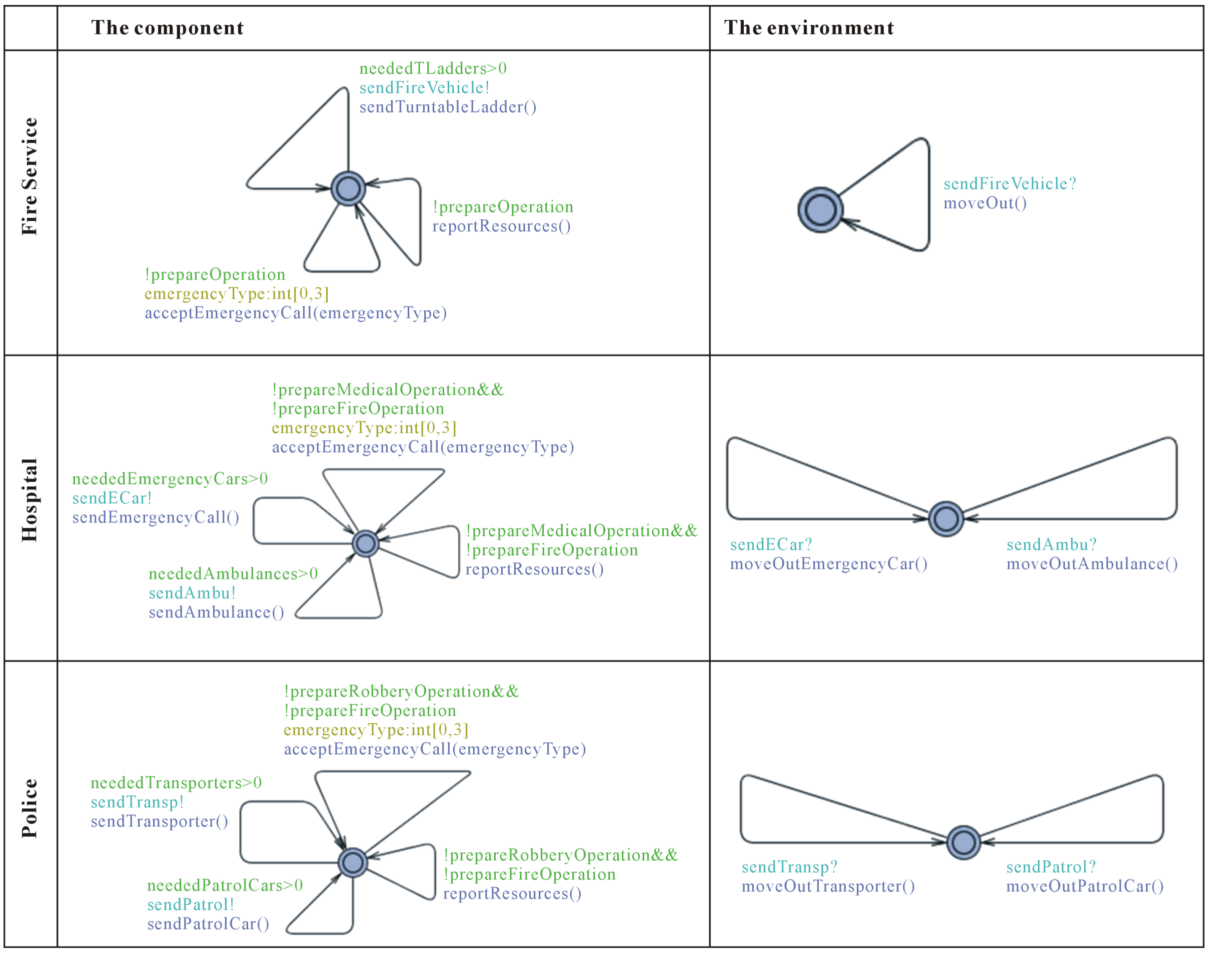

Figure 13. The headquarters and the corresponding environments as an UPPAAL automaton.

closely. First of all, each headquarters is able to transmit the current state of their own resources using the edge reportRessources. Furthermore, each of these facilities is able to use the edge acceptEmergencyCall (emergencyType) to accept an emergency call and to identify the operational type via the parameter. Finally, they can all send their respective vehicles for reported deployments (all edges with the prefix send).

Since the three organisations only require their vehicles to fulfil their duties, only these vehicles are considered in the environments. To put it more precisely, these vehicles simulate the reception of the deployment command.

Like the headquarters, the five vehicles also display very similar functions and have therefore been summarised in **Figure 14**. First, each of the vehicles is able to use the edge moveOut to leave their associated headquarters to travel to an assignment. Secondly, every vehicle is able to provide some kind of service at the location of the incident.

The assistance provided by the vehicle crews is modelled as individual edges with conditions. For example, after arriving at the incident, the crew of the fire engine must first rescue any people at risk (this happens via the edge rescuePeople). Afterwards they are able to extinguish the fire with the edge extinguishFire. It is also possible to provide assistance services more than once in the same operation. The vehicles do not maintain communication with other components and therefore do not require a modelled environment.

UPPAAL does not have more complex data types such as strings. Therefore it is necessary to encode more complex information, such as the type of operation in this scenario. In this case, the coding is kept very simple and can be understood as a type of operational code. This type of coding is also used by real operational services to ensure that the delay which occurs at the beginning of an operation is kept short. For example, the fire service often uses alarm levels or keywords, defined in the alarm and response procedure to let fire fighters know that the fire relates to a dustbin or a hazardous goods accident. Accordingly, they can make preparations on the way to their vehicles and only receive further information about the fire from headquarters once they are on their way. In this scenario, the following coding has been chosen:

0—False alarm
1—Bank robbery
2—House fire and
3—Medical emergency.

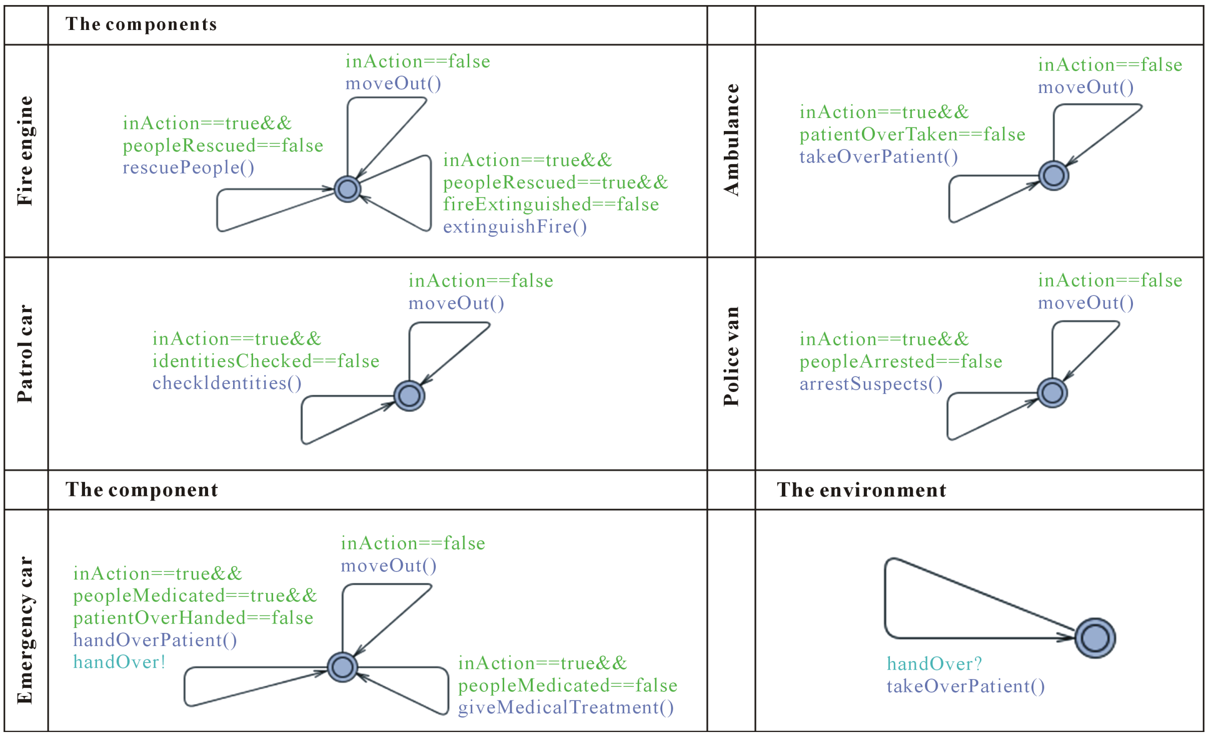

Figure 14. The vehicles and the corresponding environment as an UPPAAL automaton.

These figures apply to the entire model as an abstraction over a reported emergency. This means that the caller transmits this figure to the control centre and the control centre then decides which operational services need to be alerted.

9.5. Integration of the Components

After UPPAAL automata have been modelled of the individual components and their environments, they are connected to a single System-of-Systems with the connection tool developed in this study. A possible interconnection which is obvious based on the given scenario is shown in **Figure 15**. As already mentioned in the previous chapters, the unused components (the environments) are on the left of the diagram. On the workspace are the modelled components which stand in for the twelve automata.

The user has also already connected suitable interfaces with one another. Some interfaces of the vehicle components, such as rescuePeople (from the fire engine) or checkIdentities (from the patrol car), have not been connected deliberately, because first there are no suitable remote stations in other components and secondly these interfaces only offer services at the incident. However, it would easily be possible to model civilians, vehicles and buildings as components which would then avail themselves of these services.

9.6. Verification of the Sub-Systems and the Integrated System-of-Systems

As already mentioned the properties of certain sub-systems should also be verified in this emergency scenario. Due to the fact that operational centres only send vehicles but cannot get them back again, it is clear that these centres will no longer have any after a certain period and can therefore handle only a certain maximum number of deployments at the same time. The number of possible deployments of course depends only on how many

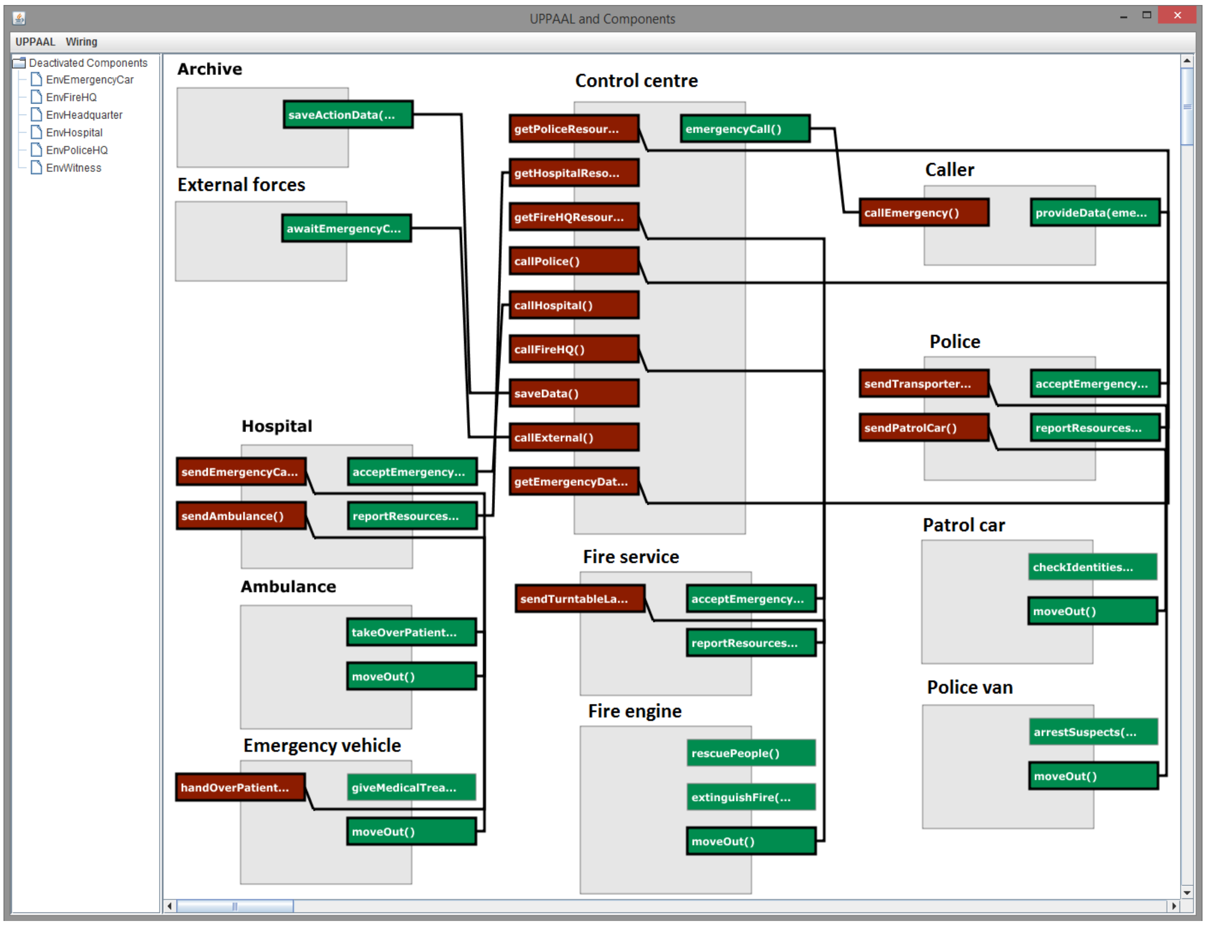

Figure 15. With ICT linked components.

vehicles of a certain type are available to the organisations. To use a simple example, the vehicles are split as follows:

- Police: Two police vans and six patrol cars
- Fire service: Four fire engines
- Hospital: Four ambulances and four emergency cars

As a reminder, the operational types and the required vehicles are listed again below (see section 9.2). In a bank robbery, three patrol cars and one police van are required. A medical emergency requires one ambulance and an emergency car. A major fire is fought with the help of two ambulances, two emergency cars, two fire engines and two patrol cars.

If the sub-systems, *i.e.* the operational centres of the three organisations, are regarded in isolation from one another, it is easy to recalculate the maximum number of emergencies that can be dealt with at the same time. The police might be able to attend a maximum of three emergencies as shown in Equation (5), if these emergencies were only fires, as in this case three batches of two patrol cars (*i.e.* six vehicles altogether) could be sent before the organisation is not able to deal with any more emergencies. If the emergencies were only bank robberies, all police vehicles would have to be deployed. The maximum number of emergencies would still only be two. Equation (6) describes that the same applies to the hospital which can handle no more than four emergencies, if there are only medical emergencies because during this type of operation the fewest number of vehicles are needed. The fire service can only be called out to deal with fires. As fires always require two fire engines, the fire service can only extinguish two fires at the same time as described by Equation (7).

To check these calculations, a variable named *active* has been introduced into the models of the three organisations, recording the number of active deployments. The maximum number of deployments can be found with the help of the UPPAAL verifier. If the following specifications (E<> in UPPAAL is the same as EF in TCTL) are applied to the respective sub-systems, these will be verified as correct.

$$E <> PoliceHQ.active \geq 3 \tag{5}$$

$$E <> Hospital.active \geq 4 \tag{6}$$

$$E <> FireHQ.active \geq 2 < 0 \tag{7}$$

If the lower values for these active deployments in the specifications were increased by only one, these would be verified as incorrect. Therefore, the mental calculation made at the outset has been confirmed by the model checker. The question that arises at this point is whether the three specifications made are still correct if the sub-systems are interconnected. This interconnection has already been carried out in the previous section. The constraints are now required and they therefore also have to be combined. As already explained in chapter 7, a combination of sub-models also results in the combination of the Equations (5), (6) and (7) for this sub-model. In this case, the constraints are combined as follows:

$$E <> PoliceHQ.active \geq 3 \bigcup E <> FireHQ.active \geq 2 \bigcup E <> Hospital.active \geq 4 \tag{8}$$

$$\Rightarrow E <> \left(PoliceHQ.active \geq 3 \bigcup FireHQ.active \geq 2 \bigcup Hospital.active \geq 4\right) \tag{9}$$

$$\Rightarrow E <> \left(\left(PoliceHQ.active + FireHQ.active + Hospital.active\right) \geq 9\right) \tag{10}$$

Equation (10) describes the constraint that was newly constructed in this way is checked with the System-of-Systems using the UPPAAL verifier. This establishes that the constraint is no longer valid, although the individual constraints were valid for the sub-systems. The reason for the failure lies in the fact that the sub-systems now have to help each other to create the deployments and this was previously not necessary. The crucial point at which the SoS fails is in the type of operation we know as fire, as a fire involves a number of organisations. The police and fire service are already able to operate the most deployments as sub-systems if the deployment relates to a fire. However, this is not the case for the hospital which reaches the maximum number of deployments only with medical emergencies. If the hospital now has to help out with fires dealt with by the fire service and police, each fire would require twice as many vehicles as medical emergencies. This inevitably leads to a reduction in the maximum number of vehicles available for deployment.

10. Conclusions and Discussion

This study has presented a new modelling technique based on networked time-based automata and shown how sub-systems can be modelled in isolation from the rest of the SoS using this technique. Sub-systems were therefore divided into software components and their expected environments. This approach has made it possible to check the constraints required for these components in terms of their accuracy without having available the remaining parts of the SoS. The study then showed how the sub-systems are integrated into a SoS. The subsequent verification of the newly constructed SoS showed, however, that although a constraint in a sub-system is correct, this does not apply to the same constraint if it is tested in the complete SoS.

Since this problem is generally very difficult to predict, this study has developed a tool which helps the developer to connect a lot of modelled sub-systems into a SoS. The aforementioned networked automata are used as a basis for the sub-systems and the resulting SoS. The interconnected SoS can then formally be verified again and the problem described can be identified and repaired very quickly.

The tool developed in this study already enables the interconnection of formally defined sub-systems into a SoS and therefore constitutes a first step towards a general formalization of these sub-systems. The easier identification of unexpected behaviour which is made possible by this methodology is an important step in making the SoS altogether more secure and reliable.

However, some facets of this type of system have deliberately been omitted, and these require further research. On the one hand, only those SoS have been considered where all sub-systems are already known at the time of development. It would be important to know at this point what happens to the validity of the constraints when components that already exist leave the system or new components enter the system, which would also mean that connections between these would change dynamically.

The autonomy of the sub-systems is another factor to which this study has only paid minor attention. The sub-systems were only able to fulfil their functionality if they were connected to the rest of the system. However, if they were separated from the rest of the system, these sub-systems were no longer able to carry out their tasks. However, if the parts of a SoS are regarded as autonomous, the SoS would still continue to fulfil its functionality despite its separation from the rest of the system, albeit to a reduced extent. The question that arises here is whether decoupled sub-systems can make a contribution with reduced functionality to preserve the constraints of a SoS in spite of all this.

Acknowledgements

We acknowledge support by Deutsche Forschungsgemeinschaft and Open Access Publishing Fund of Clausthal University of Technology.

References

[1] Szyperski, C. (2002) Component Software: Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc., Boston.

[2] Larsen, K.G., Pettersson, P. and Yi, W. (1997) Uppaal in a Nutshell. *International Journal on Software Tools for Technology Transfer*, **1**, 134-152. http://dx.doi.org/10.1007/s100090050010

[3] Clarke, E.M. and Emerson, E.A. (1982) Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In: Kozen, D., Ed., *Logics of Programs*, Springer Berlin Heidelberg, 52-71. http://dx.doi.org/10.1007/bfb0025774

[4] Kripke, S.A. (1963) Semantical Considerations on Modal Logic. *Acta Philosophica Fennica*, **16**, 83-94.

[5] Owicki, S. and Lamport, L. (1982) Proving Liveness Properties of Concurrent Programs. *ACM Transactions on Programming Languages and Systems*, **4**, 455-495. http://dx.doi.org/10.1145/357172.357178

[6] Kuhn, S.J. (1980) Modal Logic: An Introduction. xii, 295. In: Chellas, B.F., Ed, *Dialogue*: *Canadian Philosophical Review*/*Revue Canadienne de Philosophie*, Vol. 21, Cambridge University Press, New York, 545-549.

[7] Emerson, E.A. and Halpern, J.Y. (1986) "Sometimes" and "Not Never" Revisited: On Branching versus Linear Time Temporal Logic. *Journal of the ACM*, **33**, 151-178. http://dx.doi.org/10.1145/4904.4999

[8] Clarke, E.M. (2008) The Birth of Model Checking. In: Grumberg, O. and Veith, H., Eds., 25 *Years of Model Checking*, Springer Berlin Heidelberg, 1-26. http://dx.doi.org/10.1007/978-3-540-69850-0_1

[9] Clarke, E.M., Klieber, W., Nováček, M. and Zuliani, P. (2012) Model Checking and the State Explosion Problem. In: Meyer, B. and Nordio, M., Eds., *Tools for Practical Software Verification*, Springer Berlin Heidelberg, 1-30. http://dx.doi.org/10.1007/978-3-642-35746-6_1

[10] McMillan, K.L. (1993) Symbolic Model Checking. Symbolic Model Checking. Springer, 25-60.

http://dx.doi.org/10.1007/978-1-4615-3190-6_3

[11] Godefroid, P. and Wolper, P. (1992) Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. In: Larsen K.G. and Skou, A., Eds., *Computer Aided Verification*, Springer Berlin Heidelberg, 332-342. http://dx.doi.org/10.1007/3-540-55179-4_32

[12] Clarke, E., Grumberg, O., Jha, S., Lu, Y. and Veith, H. (2000) Counterexample-Guided Abstraction Refinement. In: Emerson, E.A. and Sistla, A.P., Eds., *Computer Aided Verification*, Springer Berlin Heidelberg, 154-169. http://dx.doi.org/10.1007/10722167_15

[13] Uppsala University, Aalborg University UPPAAL. www.uppaal.org