ascens

www.ascens-ist.eu

# ASCENS

## Autonomic Service-Component Ensembles

# JD3.1: Verification Results Applied to the Case Studies

Grant agreement number: **257414**
Funding Scheme: **FET Proactive**
Project Type: **Integrated Project**
Latest version of Annex I: **Version 2.2 (30.7.2011)**

Lead contractor for deliverable: **UJF-VERIMAG**
Author(s): **Jacques Combaz (UJF-Verimag), Alberto Lluch Lafuente (IMT), Ugo Montanari (UNIPI), Rosario Pugliese (UDF), Matteo Sammartino (UNIPI), Francesco Tiezzi (IMT), Andrea Vandin (IMT), Christian von Essen (UJF-Verimag)**

Reporting Period: **3**
Period covered: **October 1, 2012 to September 30, 2013**
Submission date: **November 8, 2013**
Revision: **Final**
Classification: **PU**

Project coordinator: **Martin Wirsing (LMU)**
Tel: **+49 89 2180 9154**
Fax: **+49 89 2180 9175**
E-mail: **wirsing@lmu.de**

Partners: **LMU, UNIPI, UDF, Fraunhofer, UJF-Verimag, UNIMORE, ULB, EPFL, VW, Zimory, UL, IMT, Mobsya, CUNI**

# Executive Summary

The ASCENS project addresses both verification of correctness of ensembles and validation of their performances. This is realized by using various techniques tailored to specific issues considered in the project: large number of components, adaptive behavior, uncertain environment. The proposed methods are formally founded, that is, they use explicit mathematical models representing system behavior and its interaction with the external environment. To illustrate the effectiveness our approach, we evaluated some of the verification and validation techniques considered in ASCENS by their application to the case studies of the project, which is reported in this deliverable. We had the following main results.

Statistical model-checking and stochastic models for swarms of robots permitted us to find the most efficient algorithms for implementing the rescue scenarios considered in the project. We verified a system of robots cooperating in real-time using a prototype tool implementing compositional verification for timed systems. For the Science Cloud case study, statistical model-checking was also used to demonstrate stable availability of the cloud against denial-of-service attacks when using the proposed pattern ASV+SR. We also proved that routing algorithms of `Pastry` used in the Science Cloud case study are correct, that is, they converge so that messages eventually reach their destination. We generated correct-by-construction policies enforcement mechanisms for the Science Cloud, starting from high-level and programmer-friendly specifications in FACPL. Correctness-by-construction principles were also used to synthesize controllers for an adaptive cruise control application targeting the e-Mobility case study. In addition, within the same framework the generated controllers can be checked against properties that were not considered for their design, which is useful for validating their robustness with respect to changes in their environment.

# Contents

**1 Introduction** **5**

**2 Statistical Model-Checking Applied to Swarm Robotics Scenario** **7**
2.1 Optimization of Self-Assembling Strategies . . . . . 7
2.2 Analysis of Collision Avoidance Mechanisms . . . . . 8
2.3 Design of a Deployment Scenario . . . . . 10

**3 Compositional Verification of a Robotics Scenario** **14**

**4 Quantitative Synthesis and Verification Framework** **17**
4.1 Specification . . . . . 19
4.2 Synthesis . . . . . 21
4.3 Verification . . . . . 23
4.4 Conclusion . . . . . 25

**5 Verification of Routing Procedures in Pastry** **25**
5.1 Routing in `Pastry`. . . . . . 26
5.2 The model . . . . . 26

**6 Stable Availability under Denial-of-Service Attacks through Formal Patterns** **28**

**7 Access Control, Resource Usage, and Adaptation Policies for a Cloud Scenario** **31**
7.1 A Cloud IaaS Scenario . . . . . 31
7.2 Policy-based Implementation . . . . . 32
7.3 Supporting tools . . . . . 34

**8 Conclusion** **35**

# 1 Introduction

There are several reasons for checking the design of a system before its real-life deployment. One of them is saving cost and time: finding misconceptions very early in a design flow very often shorten the iterative trial and error process required to obtain the desired properties. For some systems, it is practically inconceivable to modify them a posteriori (e.g. hardware systems, autonomous rovers for space missions). Formal guarantees can be also required prior to the deployment for safety reasons, e.g. for critical systems. Verification approaches rely on solid mathematical foundations to establish properties of a system at design time. They are model-based, that is, they represent the system (and its environment) using mathematical models. Therefore, they analyze properties under the assumption that the real system behaves according to its model.

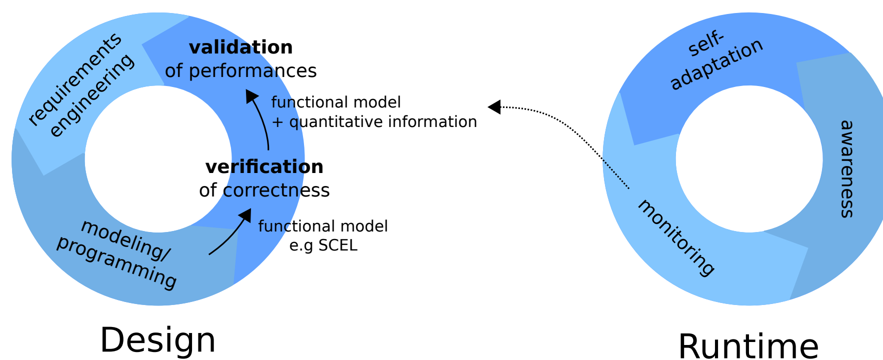



Figure 1: Verification and validation in the ASCENS Development Lifecycle.

System designers are mostly interested in two types of properties:

- *correctness* which corresponds to the fact that the system meets its requirements (timing constraints, functional requirements, etc.)
- and *efficiency* which refers to the optimality in the use of resources (time, energy, memory, etc.).

For example, amongst the correct designs, one can be more efficient than another if it uses less energy to operate. Our approach in ASCENS is to address both types of properties at design time by developing various verification and validation techniques. These techniques are tailored to the specific types of systems we consider in the project, e.g. having large number of components, including adaptive behavior, etc. Verification of correctness is based on functional models developed at design time, e.g. SCEL specifications. In a second step, these models are enriched by quantitative information to validate system performances. Notice that such quantitative models can be refined lately after system deployment to include values derived from the monitoring of real executions, but we do not provide any example of that here as validation principles remain unchanged by such refinement (see Figure 1). More details about the ASCENS Development Lifecycle can be found in Deliverable JD3.2. To illustrate the effectiveness of the verification and validation techniques proposed for ASCENS, we evaluated some of them on the case studies of the project, which is reported in this deliverable.

**Establishing correctness.** We developed compositional verification techniques for proving *safety* properties, that is, for establishing that the system cannot reach undesirable configurations, e.g. deadlock-freedom, mutual exclusion, etc. One noticeable feature of this technique is the fact that it establishes global properties of the whole system based local characterizations of its components

and their interactions, which allows to scale up to large systems. In addition to pure functional behavior, the proposed method takes into account timing constraints of components. These are interesting for verifying components coordinations which are very often implemented using protocols involving time. We present a practical use case consisting of collaborating robots and showing the applicability of the method.

For cloud computing applications, correctness of networking services includes not only the fact that messages are routed towards the right destination, but also that the target destination is eventually reached, which cannot be captured by safety properties. We applied a framework based on $\pi$-calculus to prove the convergence of routing algorithms of `Pastry` used in the Science Cloud case study.

Another way for building correct systems is to obtain correctness by construction. Such approach was followed for enforcing policies for access control, resource usage and adaptation policies, in the cloud. In the proposed methodology, the designer of the cloud write FACPL specifications allowing direct expression of policies in a programmer-friendly language. An Eclipse-based tool automatically produces Java code implementing the FACPL policies. It is based translation rules formally defined on the semantics of FACPL.

**Evaluating efficiency.** Once correctness of a design is stated, one can be interested in how it performs. Performance is strongly related to the environment in which the system is immersed: a design may perform well in some contexts, but poorly in others. Models for uncertain environments needed for the project should include ranges of possible environment behaviors that the system may face. Including probabilities permits us to quantify the degrees of likelihood of the different scenarios.

We used such stochastic models to optimize swarm behavior of the robotics case study, e.g. strategies for self-assembly or for the deployment in a complex environment. To this end, statistical model-checkers were able to compute reliable guarantees of swarm performance at reasonable cost. Indeed, by the means of partial system state coverage, they provide estimates of system parameters given a level of confidence (i.e. the probability for the computed values to be incorrect), instead of computing them exactly.

For the Science Cloud, we proposed and formally defined the design pattern ASV$^+$SR combining well known policies ASV (Adaptive Selective Verification) and SR (Server Replicator). Analysis with the statistical model-checker PVESTA shown that ASV$^+$SR ensures stability and availability of the cloud against denial-of-service attacks, which cannot be guaranteed by policies ASV and SR used alone.

We generated adaptive cruise controllers for the e-Mobility case study from Markov Decision Processes. One noteworthy feature of the proposed approach is the possibility to do control synthesis and validation/verification in concert within the same framework. This allowed us to check properties for the controlled systems (e.g. stability) even under environmental conditions for which the controller was not designed.

**Structure of the deliverable.** The rest of the deliverable is structured as follows. Section 2 shows how statistical model-checking techniques and corresponding tools has been used for optimizing and tuning algorithms implementing part of the robotics scenario proposed in ASCENS. We verified safety properties for a real-time protocol involving heterogeneous robots cooperating to acquire knowledge in Section 3. In Section 4 we present the application of control synthesis to an adaptive cruise control application for the e-Mobility case study. Sections 5, 6 and 7 target the Science Cloud case study. We verified existing routing algorithms of `Pastry` (Section 5) and stable availability of the pattern ASV$^+$SR (Section 6). We also generated of correct-by-construction code for access control policies from formal specifications (Section 7). Finally, Section 8 provides conclusion and future work.

# 2 Statistical Model-Checking Applied to Swarm Robotics Scenario

Swarm robotics systems favor the use of large number of simple robots coordinating to collectively accomplish complex tasks. The lack of individual capabilities of each robot of a swarm is counterbalanced by their cooperation abilities. Having large number of nodes is desirable whenever the risk of failure of each individual robot is high (e.g. evolving in difficult and uncertain environments): the swarm can still operate even if few robots have failed. There are so many sources of uncertainty (physical environment, hardware, sensors noise, etc.) that it is hopeless to have exact predictions of the behavior a swarm. Anyhow, swarm roboticians are usually more interested by the likely behavior of a swarm than by corner cases behavior which have very low probability to happen. They need quantitative analysis to evaluate the performances of a given design, e.g. the probability for the swarm to accomplish the target goal, the average failure rate of the robots, the expected energy consumption, etc. Statistical-model checking (SMC) is a "verification-inspired" approach that performs quantitative analyses of a system. Like model-checking, it uses formally defined models from which it explores the reachable states. In contrast to standard model-checking, SMC does not require exhaustive computation of the reachable states to conclude about a given property. It answers quantitative questions based on partial state space coverage, and evaluate *confidence* in such results thanks to stochastic models. SMC tools usually stop the analysis when the desired degree of confidence (provided as an input parameter) is reached.

We applied statistical model-checking tools to the rescue scenarios of ASCENS, which are presented in details in Deliverable D7.3. Firstly, the tool MESSI permitted us to optimize self-assembly strategies which are needed in the rescue scenarios for crossing holes and grab/pull heavy objects. Second, to integrate capabilities of MESSI in the ASCENS tool-chain, we developed the tool MISSCEL which implement analysis techniques of MESSI but for SCEL specifications. We demonstrated its applicability by considering collision avoidance strategies which are required by ASCENS scenarios. Finally, we built faithful models for the whole deployment phase of Rescue Scenario 2 (see Deliverable D7.3), and shown how the tool SMC-BIP helped us in the design of the swarm.

## 2.1 Optimization of Self-Assembling Strategies

In [BCG$^+$12b, BCG$^+$12c], we investigated a methodology to specify and analyze adaptive systems in Maude [CDE$^+$07]. The distinguishing features of this approach are: (i) a white-box approach to adaptation based on the notion of control data [BCG$^+$12a]; (ii) a hierarchical architecture to modularize the design; (iii) computational reflection as the main adaptation mechanism; (iv) probabilistic rule-based specifications and quantitative verification techniques to specify and analyze the adaptation logic. The approach can be naturally realized in Maude, a framework based on Rewriting Logic, since (i) Maude's algebraic approach facilitates the formalization of control data; (ii) hierarchical architectures such as the Reflective Russian Dolls model have been promoted and shown to be suitable to specify adaptive systems in Maude; (iii) Maude efficiently supports computational reflection; (iv) Maude is a rule-based language for which probabilistic extensions [AMS06] and analysis techniques are available. Of course, this approach can be realized in any other framework providing the necessary support for (i)–(iv).

We focused on self-assembly strategies (e.g. [OGCD10]), a mechanism allowing groups of simple entities to act as a single complex entity exhibiting emergent behaviors. We realized our approach in the MESSI tool [MES], a Maude instantiation of our methodology which allows us: (1) to model self-assembly strategies, (2) to debug them via animated simulations, and (3) to estimate their quantitative properties resorting to PVeStA, a distributed statistical analyzer and statistical model checker [AM11, SVA05], or to the recently proposed MultiVeStA [SV], which extends PVeStA. In particular, we discussed how to implement one of the robotic self-assembly strategies described

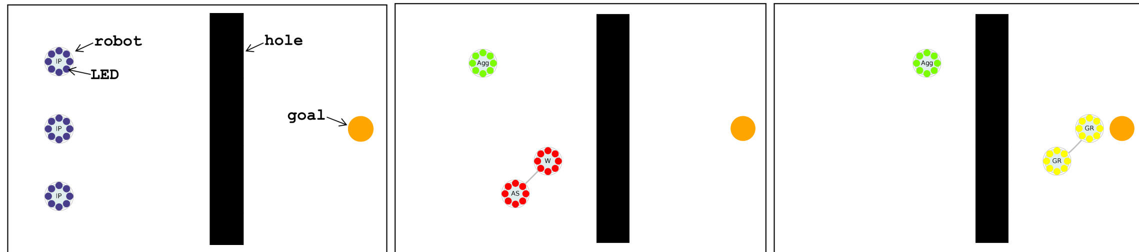



Figure 2: Three states of a simulation of self-assembling robots: initial, assembly, final.

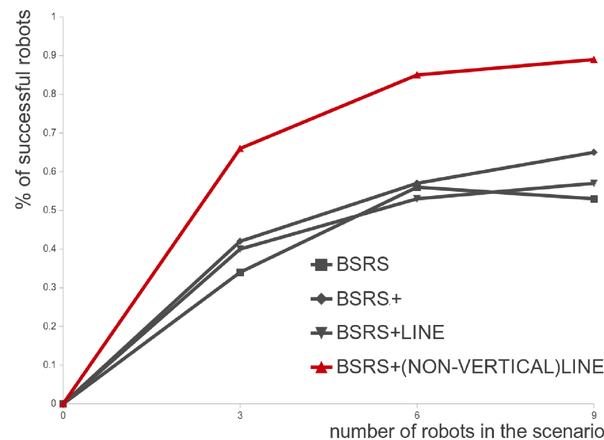



Figure 3: Ratio of successful robots.

in [OGCD10] (named *basic self-assembly response strategy*), where robots assemble to cross obstacles (like holes or hills), we proposed several variants of the strategy, and we resorted to PVeStA to study and compare the performance of the different strategies.

Fig. 2 illustrates three states of a simulation where robots execute the basic self-assembly response strategy. The initial state (left) consists of three robots (grey circles with small dots on their perimeter) in their initial state (emitting blue light), a wide vertical hole (the black rectangle) and the target of the robots, i.e. a light source (the orange circle to the right). After some steps, two robots finally get assembled (middle of Figure 2), and can safely cross the hole (right of Figure 2), while the third one remains in the left part of the arena.

As an example of the performed statistical analysis tasks, Figure 3 depicts the ratio of robots which successfully cross a hole and reach their target when varying the number of robots. We considered four different self-assembly strategies, whose details can be found in [BCG+12b, BCG+12c]. The two strategies BSRS and BSRS+ do not force robots to form particular shapes, while when using BSRS+LINE and BSRS+(NON-VERTICAL)LINE, robots try to form lines. The difference between the last two is that the latter, which outperforms all the others, does not allow vertical lines (i.e. lines parallel to the hole). Interestingly, what we found is that the extra time spent in forming lines pays back if we manage to obtain lines that are not parallel to the hole.

## 2.2 Analysis of Collision Avoidance Mechanisms

In a work related to the research line regarding the tool MESSI (discussed in Section 2.1), we focused on the use of SCEL to specify adaptive systems, and to the development of tools to analyze SCEL specifications. The result is MISSCEL: a Maude Interpreter and Simulator for SCEL. Indeed, the research line concerning MISSCEL can be seen as a sort of follow up of the more prototypal and early

stage one concerning MESSI.

The SCEL language comes with solid semantics foundations laying the basis for formal reasoning. MISSCEL, a rewriting logic-based implementation of SCEL's operational semantics is a first step in this direction. MISSCEL is written in Maude, which allows to execute rewrite theories; what we obtain is then an executable operational semantics for SCEL, that is an interpreter. Given a SCEL specification, thanks to MISSCEL it is possible to exploit the rich Maude toolset [CDE+07] to perform: (i) automatic state-space generation; (ii) qualitative analysis via Maude's invariant and LTL model checkers; (iii) debugging via probabilistic simulations and animations generation; (iv) statistical quantitative analysis via the recently proposed MultiVeStA [SV], a distributed statistical analyzer extending PVeStA [AM11, SVA05].

A further advantage of MISSCEL is that SCEL specifications can now be intertwined with raw Maude code, exploiting its great expressiveness. This allows to obtain cleaner specifications in which SCEL is used to model behaviors, aggregations, and knowledge manipulation, leaving scenario-specific physical details like environment sensing abstractions or robots movements to Maude.

Noteworthy, in [BDVW] we have shown how to enrich SCEL components with reasoning capabilities exploiting the reasoner Pirlo [Bel13], implemented as well in Maude, and we have analyzed a collision-avoidance robotic scenario. The scenario is concerned with a group of robots moving in an arena, abstracted as a discrete grid. Robots are situated in cells intersections, and perform one-cell movements (up, right, down or left), or stay idle. We considered two kinds of robots distinguished by how the choice of action is performed: *normal robots*, and *informed robots*. Normal robots randomly choose the action to be executed among the five listed above, i.e. they perform a random walking or stay idle. The informed robots monitor their surrounding environment by relying on proximity sensors, and exploit this information to choose actions aiming at reducing the number of collisions. The amount of environment perceived by an informed robot depends on its perception range. The positions up, right, down and left are reachable with a single move, while the diagonal ones are reachable with two moves. However, the perception of the diagonal positions is also useful for the computation of the next action, as a robot in there (e.g. one perceived in down-left) could move towards the same position chosen by the informed robot (e.g. up, if the informed robot moves left).

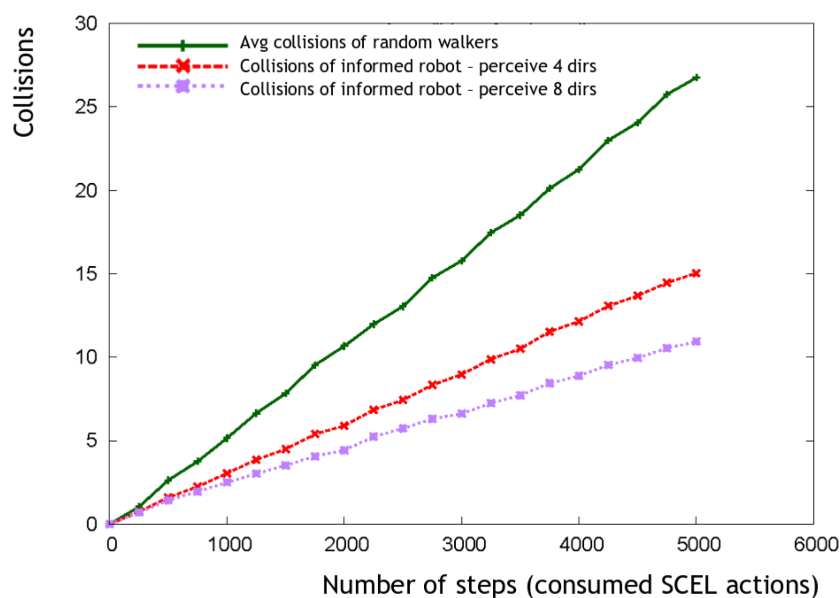



Figure 4: Collisions of normal and informed robots at varying of the number of steps.

In our analysis we considered two scenarios with ten normal robots and an informed one, varying the size of the perception range of the informed robot. In the first scenario the informed robot perceives only the four surrounding positions (up, right, down, left). In the second scenario the informed robot has a wider perception range, allowing to perceive also the positions in the four diagonal directions (up-right, down-right, down-left, up-left). For both scenarios we first studied the expected value of the average number of collisions of the normal robots when varying of the number of execution steps. Not surprisingly, we obtained very similar measures for both the scenarios, and hence we use only one plot in Figure 4 ("Avg collisions of random walkers"). More interesting is the case of informed robots. As depicted by the plots "Collisions of informed robot - perceive 4 dirs" and "Collisions of informed robot - perceive 8 dirs", informed robots do significantly less collisions than the normal ones, and wider perception ranges allow to further decrease the number of collisions.

## 2.3 Design of a Deployment Scenario

We applied the statistical model-checking tool SMC-BIP to the robotics case study of ASCENS. The choice of BIP as a modeling language for the scenario is motivated by the fact that it can also be the starting point for verification [BBNS10] and code generation [BBB+11]. We focused on Scenario 2 of the robotics case study, which is extensively described in Deliverable D7.3. It consists in (1) deploying a swarm of marXbot [BLM+10] robots—the explorers—to find victims (which are other marXbots) distributed all over an arena shown in Figure 5, and (2) to rescue the victims. We restricted our analyses to the deployment phase only, and we also adopted the following simplifications with respect to the description provided in D7.3:

- robots stop only if they are close to victims or far away from other landmarks robots (i.e. no detection of corridors which corresponds to stopping condition *(i)* of D7.3)
- we do not build any routing tables, and landmark robots simply route robots backwards if no further exploration is needed.

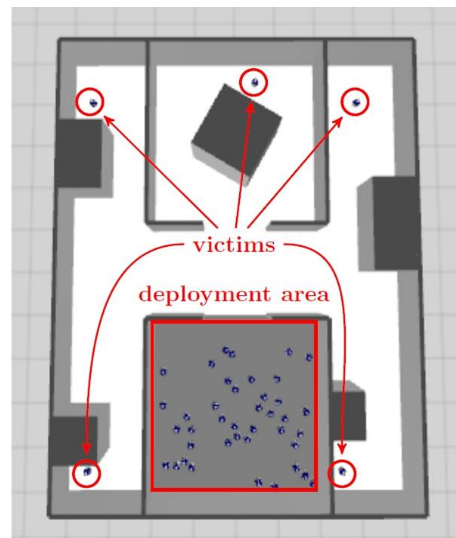



Figure 5: Arena of the rescue scenario.

We first built a BIP model of the marXbot based the description found in D7.3. It includes a faithful implementation of the 24 proximity sensors as well as the rotating scanner of the robot, considering noisy values for all the sensors. To keep our model simple, we used abstractions for representing the detection of landmarks by the camera and the communications through the range-and-bearing module. The skeleton of the BIP component used for modelling the marXbot behavior is provided in Figure 7. Following the approach implemented in the simulator ARGoS [PTO+12], we rely on synchronous discrete time execution with a duration of 10 ms for the time steps. This is implemented by a connector (*tick*) synchronizing all the robots (victims and explorer) as shown in Figure 6. Notice that we also stop the execution when all the victims are found, by disabling connector *tick*. The model of the swarm represents 1500 lines of BIP code along with 1200 lines of external C++ code.

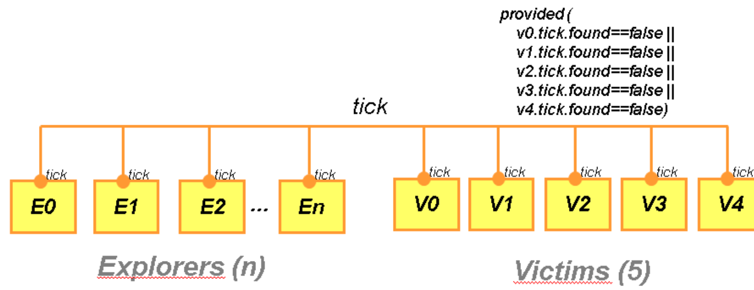



Figure 6: Architecture of the BIP model built for the whole system.

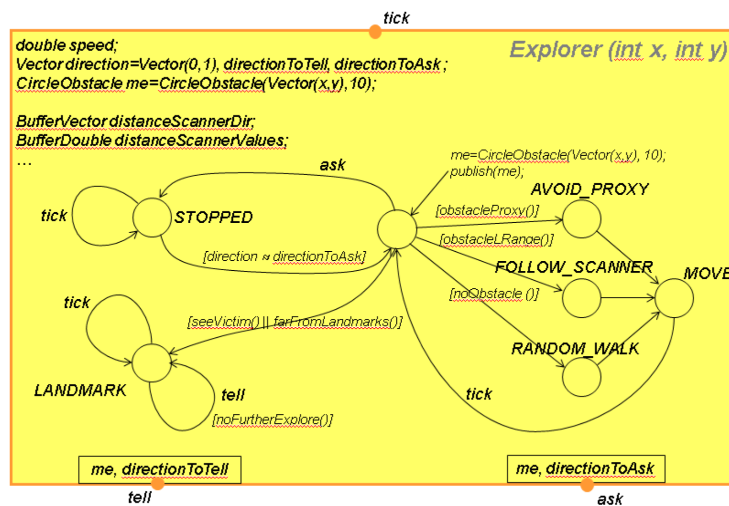
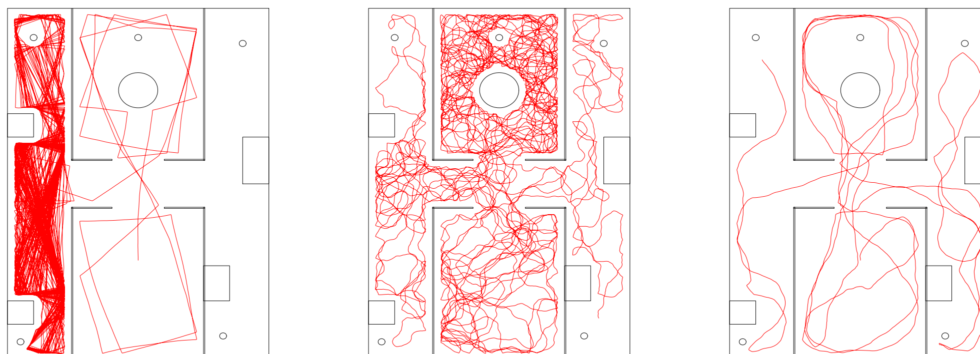



Figure 7: BIP model for the behavior of a single robot.



(a) straight (b) random (c) random + scanner

Figure 8: Simulation of a single robot and various moving strategies.

| strategy: | straight | random | random + scanner | | | |
|---|---|---|---|---|---|---|
| number of explorers: | 1 | 1 | 1 | 11 | 21 | 31 |
| 1st victim | 207 | 1243 | 556 | 175 | 149 | 154 |
| 2nd victim | 2265 | 5619 | 790 | 329 | 220 | 176 |
| 3rd victim | 2983 | 12675 | 1231 | 551 | 481 | 273 |
| 4th victim | timeout | 16053 | 3075 | 964 | 638 | 500 |
| 5th victim | timeout | 16883 | 3358 | 1134 | 645 | 540 |

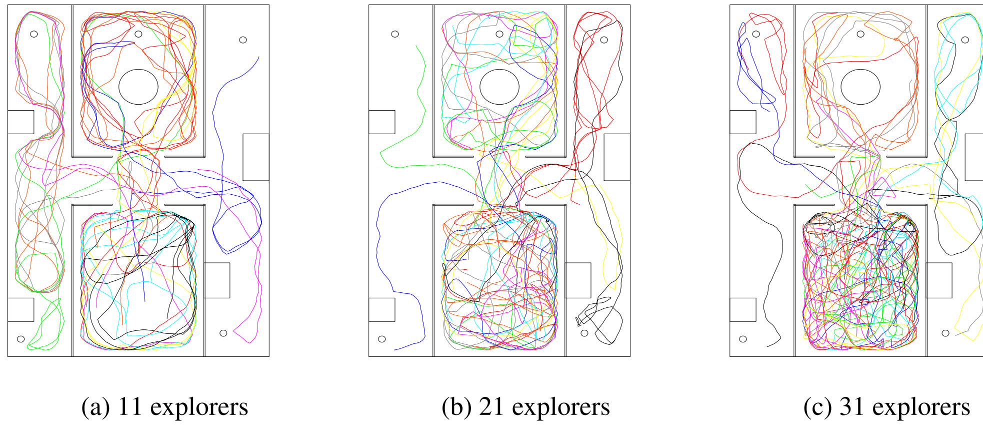
Figure 9: Delays in seconds for finding victims corresponding to simulations of Figures 8 and 11.

| strategy: | straight | random | random + scanner | | | |
|---|---|---|---|---|---|---|
| number of explorers: | 1 | 1 | 1 | 11 | 21 | 31 |
| 1st victim ($\alpha$=$\beta$=$\delta$=0.05) | 343 | 2996 | 892 | 211 | 188 | 152 |
| 5th victim ($\alpha$=$\beta$=$\delta$=0.01) | timeout | 41250 | 11562 | 1171 | 820 | 742 |

Figure 10: Delays in seconds computed by SMC-BIP for finding victims with probability $P$=0.85.

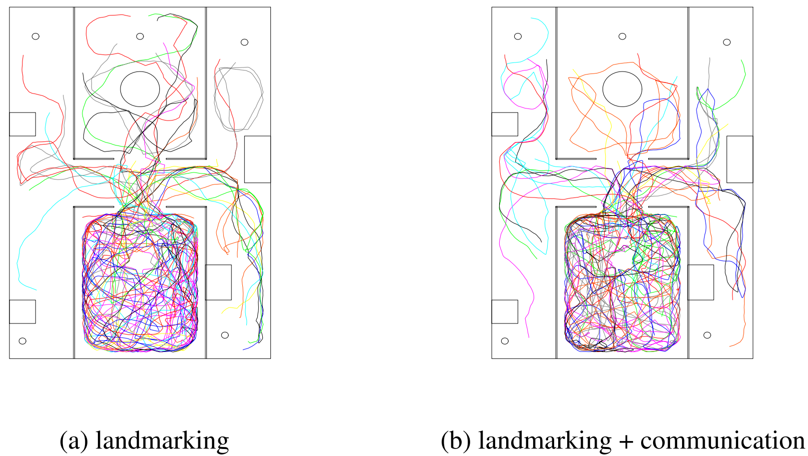
**Single robot behavior.** We started by experimenting with several behavioral strategies for a single robot: straight walk, random walk, and random walk improved using the rotating scanner. All includes basic obstacle avoidance so as not to bump into walls and/or other robots. Figures 8 and 9 shows examples of simulations obtained for the different strategies and the corresponding delays for finding the victims. In Figure 8, victims are the five small circles (three at the top and two at bottom) in the arena, and the path followed by the explorer is represented by the red drawing. Using straight walk minimize the distance for travelling from one location to another. However, it resulted in a very poor coverage since the explorer was trapped on the left side of the arena from which it did not escape even after a long time. Random walk led to good coverage but longer delays for finding the first three victims than the ones obtained with straight walk. From this observation, we improved random walk by using the rotating scanner which allows the explorer to track long distances obstacles and to follow corridors and walls, which is clearly visible on simulations (see Figures 8 and 9). All these observations are confirmed by the analysis performed by SMC-BIP with which we computed the expected time for finding the 1st and the 5th considering probability 0.85, provided in Figure 13. Parameters $\alpha$, $\beta$ and $\delta$ in table of Figure 13 correspond to the target degree of confidence for SMC-BIP. The lower these parameters are, the lower the probability to obtain an incorrect answer is. They are formally defined in [BBD$^+$12]. Using SMC-BIP we also managed to show that increasing the number of explorers (we tested for 11, 21, and then 31) tends to reduce the expected delays for finding victims (see Figure 13). Examples of simulations traces for 11, 21, and 31 explorers can be found in Figure 11.

**Cooperation between robots.** We completed the model by including part of the cooperating behavior of the scenario proposed in D7.3. First, we added landmarking behavior: if a robot become too far away from other landmarks, or if it finds a victim, it stops to establish a new landmark. The goal of these landmarks is to avoid exploring areas that have been already explored. Landmarking alone reduced drastically the performances, as shown by Figure 13. This can be explained by the fact that landmarking reduces the moving range of the explorers and decreases the number of active robots, sometimes to the point where all robots were stopped (i.e. were landmarks) whereas victims remained to be found. An example of such situation can be observed in Figure 12a.



(a) 11 explorers (b) 21 explorers (c) 31 explorers

Figure 11: Simulation of random + scanner strategy for different sizes of swarms.



(a) landmarking (b) landmarking + communication

Figure 12: Simulation of landmarking strategies for 31 explorers.

| **strategy:** | landmarking | landmarking + communication |
|---|---|---|
| 1st victim ($\alpha$=$\beta$=$\delta$=0.05) | ? | 375 |
| 5th victim ($\alpha$=$\beta$=$\delta$=0.01) | timeout | 1797 |

Figure 13: Delays in seconds computed by SMC-BIP for finding victims with probability $P$=0.85.

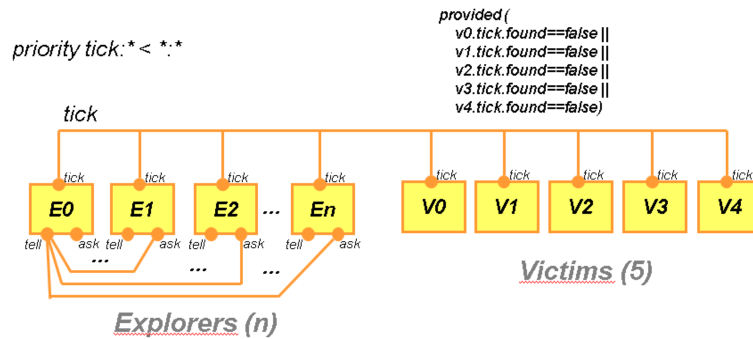



Figure 14: Architecture of the BIP model including communications explorers and landmarks.

The goal of landmarking is mainly to reduce the time to accomplish the second phase of the scenario. To this purpose, landmarks must communicate with active robots to route them for achieving their goal (exploring, rescuing, etc.). We included basic communication capabilities in the model allowing landmarks to route back robots if there is no need for exploration in their given direction (e.g. presence of a dead end). These communications were implemented by simple connectors between the robots (see Figure 14). By the way, this shows limitations of (static) BIP representations as we had to include all possible connections between the robots, that is, $n^2$ connectors when using $n$ explorers. It would have been better to use dynamic description of connectors as it is possible with DyBIP [BJMS12], but this feature was not part of the existing tool-chain we used. Adding communications allowed acceptable performances for finding all the five victims, while establishing landmarks required by the second phase of the scenario. Simulation traces clearly show the switch-backs performed by the robots when meeting landmarks from which no further exploration is needed (see Figure 12b).

SMC-BIP allowed us to fine-tune the behavior of the marXbot to optimize the deployment phase of the ASCENS scenario. Such fine-tuning is also possible with standard simulation techniques (e.g. with ARGoS), but statistical model-checking permits us to have reliable information about the performances of the swarm, guaranteed by explicit degrees of confidence and based on exploration of possible behaviors. For example, it required sometimes more than 20000 simulations for SMC-BIP to conclude on a single delay value. The BIP model we developed can also be a basis for computing stochastic abstractions and/or for applying verification techniques and tools.

# 3 Compositional Verification of a Robotics Scenario

We applied the compositional verification techniques for timed systems presented in Deliverable D5.3 to a robotics scenario of ASCENS partner EPFL. It consists of cooperating robots used in a child's bedroom for home automation, automatic cleaning, or child assistance in tidying up. We considered the following types of robots/devices in the room, all capable of wireless communications.

**Cleaning Robot.** We assume the presence of an autonomous vacuum cleaner. This kind of domestic robot is nowadays widely used and working well, e.g. Roomba Vacuum Cleaning Robot. The distinguishing feature we consider in our setting is its ability to cooperate with other types of robots.

**Toy Case Robot.** The toy case robot—called Ranger—is currently developed in a research project of EPFL [NCC]. Its goal is to encourage the child to put away the toys in the case. To this end,

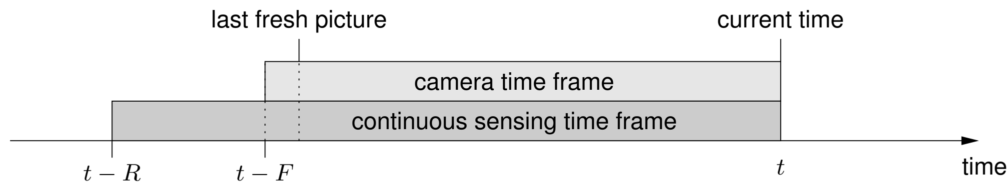



Figure 15: Freshness parameters $F$ for the camera and $R$ for the other devices.

it interacts with the child and produces pleasing sound and light each time the child takes toys from the floor and puts them in the case. The effectiveness of such robot has already be shown in previous studies. We also assume that this robot has sensors able to detect the presence of the child when he is close enough.

**Bed and desk chair.** We assume that the bed and the desk chair are equipped with sensors allowing to detect when the child seats on.

**Door.** We assume that the bedroom door is equipped with an electric closing and locking system. A safety mechanism stops any closing procedure if the child tries to enter the bedroom while the door is closing.

**Ceiling Camera.** A camera located on the ceiling can take pictures that can be analyzed to detect whether the child is in the bedroom. The shape of the child can be tracked in these pictures only if it is not too close to other shapes, i.e. if the child is not playing with the toy case and not on the bed or the chair. For energy consumption reasons, we assume that the pictures are shot and analyzed only once a while, e.g. at a given period $P$.

In this scenario we were interested in different properties. The first one is a safety property stating that the child should not be in the bedroom while the cleaning robot is cleaning. To this end, we designed a protocol in which the cleaning robot (1) checks if child is outside the bedroom by correlating information from all the other robots / devices, (2) if so, closes and locks the door to keep the child outside, and (3) cleans the bedroom. We used formal verification to prove that our protocol satisfies the safety property. Other properties of interest are quantitative, such as the average time between consecutive cleanings or the average number of toys in the bedroom, but they are not part of the study presented here.

The first thing one can observe in this system is that knowledge—e.g. the presence of the child—is distributed amongst the robots. One major issue for the cleaning robot is to build a consistent view of the status of the child (inside or outside) from local knowledge of the robots, and all this in real-time. We assume continuous sensing of the child for the case, the bed and the chair. On the other hand, pictures are taken only at specific time instants meaning that we have to deal with outdated information for the camera. If the child is not on a picture taken at a given time, then it was either outside, or inside and playing with the case, or on the bed or the chair. If we want to be sure that the child was outside the bedroom at the time the picture was taken, we need to know what was the status of the sensors of the case, the bed and the chair at this time. For this purpose we associate one timer to each sensor and reset it each time the child leaves. We also used a freshness parameter $F$ for controlling the knowledge of the camera: the child is considered outside by the camera if he was not in a picture taken at most $F$ time units ago. In a slightly different way, we used parameter $R$ for the case, the bed and the chair: the child is considered outside by these devices if he was not detected for at least $R$ time units. Notice that if $R \geq F$ we can safely conclude whether the child was outside or

inside at the time the last picture was shot from the camera (see Figure 15). We also use $R$ for the door, that is, it is considered closed if it was closed for more than $R$ time units.

We built a BIP model for verifying the principles of the proposed protocol at high level (see Figure 16). If the child is not detected by all the devices[1] (w.r.t. $F$ and $R$), the cleaning robot starts locking the door since there is a high probability that the child is not in the bedroom at the current time (we are sure that at some instant in the last $F$ time units, the child was not in the bedroom). This is represented by a strong synchronization between ports collab, close and noChild (the yellow ports of Figure 16). Notice that the behavior for parameters $F$ and $R$ is ensured by local conditions based on components clocks. Once executed, the door starts closing, and the case and the chair move towards locations that ease the cleaning robot to operate (under the bed for the case, and next to the desk for the chair). Then the cleaning robot starts cleaning only if the child is still not detected by the devices and the door is still closed, considering again parameters $F$ and $R$. If so, it locks the door and starts cleaning, which is modelled by a strong synchronization between ports startClean, lock, underBed, reachedDesk, noChild, noChildP (the green ports of Figure 16). Otherwise, if the cleaning is not possible for 120 time units, the cleaning robot timeouts and returns to its initial state. Intuitively this protocol is safe (provided $R \geq F$) since the cleaning starts only if the child was outside when the last picture was taken and the door was kept closed since this time. Moreover, during cleaning, the door remains closed by using the locking mechanism.

Using our verification technique for timed systems (see Deliverable D5.3) we managed to prove formally that the child is not in the bedroom while the robot is cleaning, for any value of the parameters such that $R \geq F$. More precisely, if the cleaning robot is in control state C, then the child is in state 0 (these control states are in blue in Figure 16). This property is non trivial as it strongly depends on the individual behavior of all the devices and in particular their timings, and it can be tricky to ensure for the system. When we developed this case study, we experienced this fact and and did several attempts before we obtained a correct design. Verification tools helped us in finding and fixing erroneous behaviors. In fact, when the tool was not able to prove the correctness of the system, it exhibited a potential[2] execution sequence violating the property which was used to fix the protocol. Notice that the model proposed here is far to abstract to be used directly for implementing the devices. It uses primitives such as atomic synchronizations between two or more components (i.e. multi-party interactions) that should be translated into simpler interactions (e.g. messages passing). To get correct-by-construction implementations we could transform the proposed BIP model into a Send/Receive BIP model using techniques developed for generating distributed implementations from BIP, as presented last year in D2.2 for untimed systems. We are currently extending such approaches to take into account timing constraints. Another working direction is to enrich the model with quantitative information (e.g. energy consumption, stochastic model of the child, etc.) and use formal verification and/or control synthesis approaches to find optimal values for parameters $P$, $F$ and $R$.

# 4 Quantitative Synthesis and Verification Framework

We developed a framework that integrates various recent and more classic algorithms for quantitative verification and synthesis. This concerns partly Task T5.1: Verification and Design of Service Components "Model checking and Synthesis in a quantitative Framework." To our best knowledge, this is the first time that verification and synthesis work in tandem in the same framework. Figure 17 gives a general overview of our framework.

[1] Actually the complete model has additional conditions such as the number of toys in the case, but they have no impact on the safety property, so we do not consider them here.

[2] The execution sequence is only potential since the method relies on over-approximated system behavior, and hence it can point out problems that cannot occur in the actual behavior.

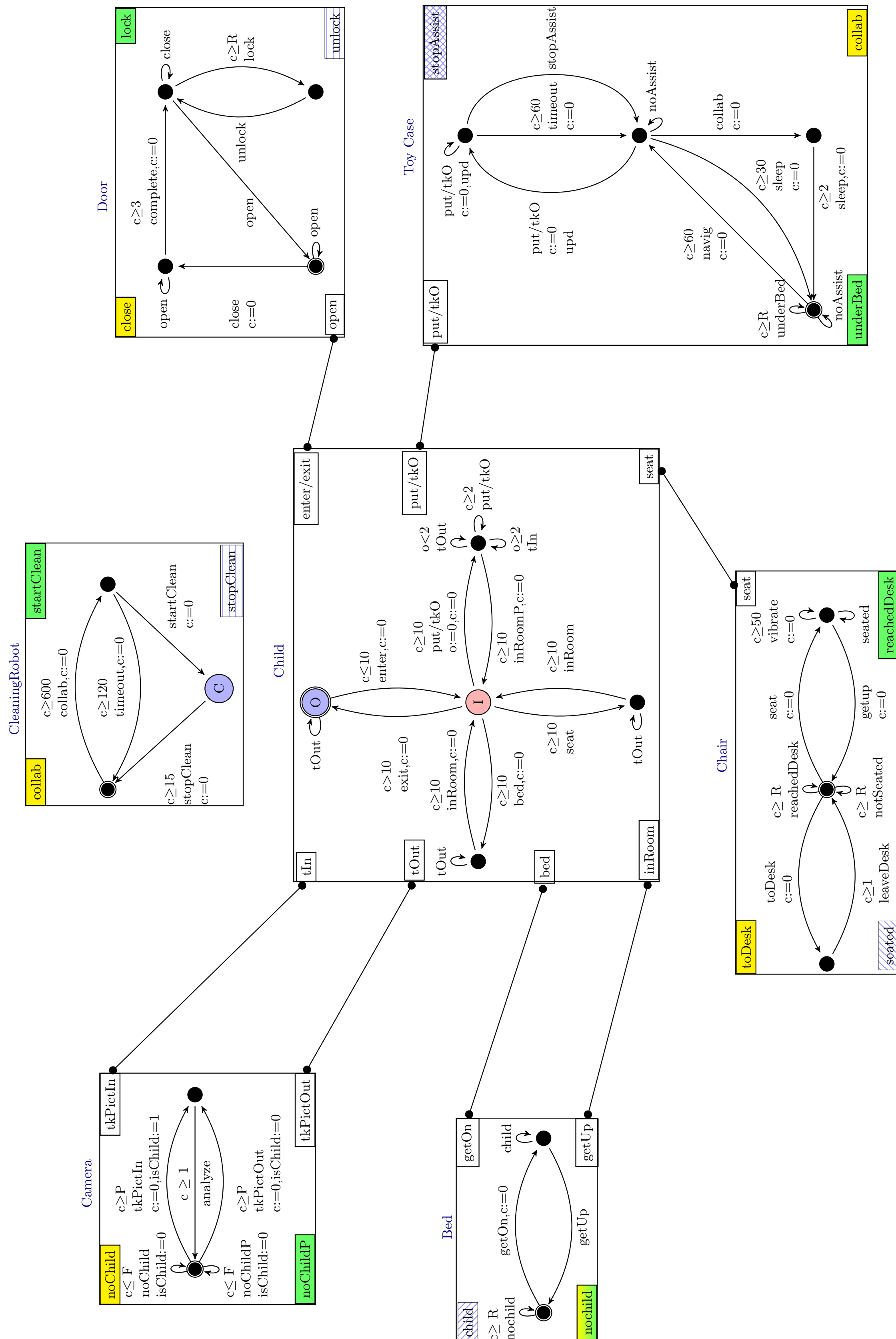


Figure 16: BIP model of the cooperating robots example.

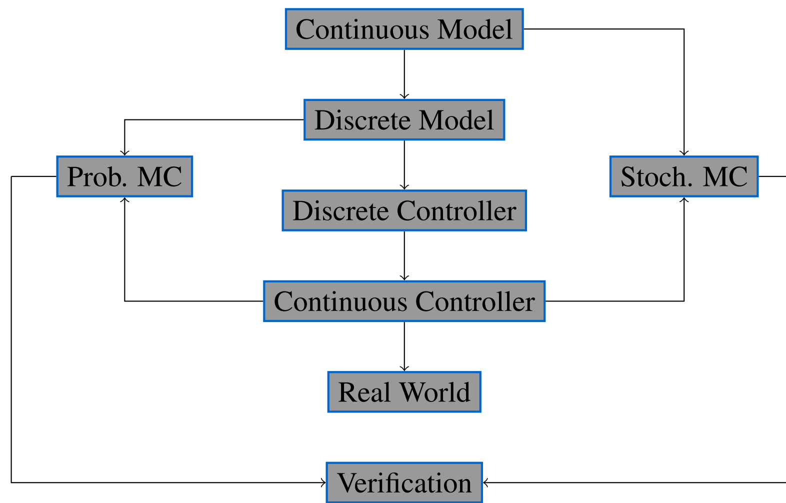



Figure 17: Framework Overview

At the core lies a Java Embedded Domain Specific Language for describing continuous models. We provide, hidden from the user, means for deriving a discretized from the continuous model. The user has full access to this derived model and can use it in his own programs. Our framework provides several algorithms to effectively find optimal controllers for the discretized model. We also provide several ways to use these discretized controllers as controllers in the original continuous model.

In addition we provide two means of controller validation and verification. First, Probabilistic Model Checking checks the performance of the continuous controller against a discretized model. This discretized model does not have to be (and should not be) the same as the discretized model from which we generated the controller. Instead, we should use different parameters for discretization as well as different parameters in the model itself to check the model for robustness and over-fitting. Second, we have implemented Bayesian Stochastic Model checking which allows to validate and verify the controller in a continuous environment. This allows us to check the controller against models with different parameters, as well as to check errors induced by the discretization.

For an example of a controller our system generates and validates, consider an Adaptive Cruise Control (ACC) system. ACCs are now built into luxury cars and are responsible for automatically maintaining a fixed distance to the car in front. Such a system senses (1) the current distance between the car it equips and the car in front, and (2) their relative velocity, i.e,. by how much the distance shrinks or grows per second. On the one hand, the goal of this system is to reach and maintain the desired distance quickly. On the other hand, the controller is also responsible for pleasant driving. That is, it should not unnecessarily or suddenly accelerate or jerk (where jerk is the change of acceleration over time). There is an obvious trade-off between these two criteria, and our framework allows study of trade-offs like these. An additional concern is that the relative velocity is not exclusively under the control of the system. Instead, since we do not know and cannot predict the other driver's intentions, we assume that she is going to behave randomly.

We support various algorithms for finding such optimal controllers, and we also support analyzing the controller we generate. The analysis is necessary for the following reasons.

1. Controllers are generated under certain assumptions; we need to know what happens if these assumptions are not met by the real world.

2. Controllers are optimized with regard to a certain set of criteria; we might be interested in other

criteria as well.

3. Continuous state space makes it necessary that we use discretization techniques; we have to study the influence discretization has on the controller.

4. Behavior of the environment is modelled as probabilistic; we want to know the worst-case behavior of the controller under all possible behaviors of the environment.

Formally, we use Markov Decision Processes (MDPs) as models. A Markov Decision Process is a probabilistic graphical model with a finite set of states $S$. In each state, there is at least one active action from the finite set of possible actions $A$. Function $\alpha : S \to 2^A \setminus \{\emptyset\}$, defines what actions are active in what state. For each pair of states and active actions, there is a probability distribution defining how likely it is that we go to some state from a state with that action, i.e., $p : S \times A \to D(S)$. Lastly, we want to evaluate the costs/rewards of each decision. To that end, we define a vector of functions $R = (S \to \mathbb{R})^n$. A MDP is then a tuple $M = (S, A, \alpha, p, R)$.

## 4.1 Specification

Our framework supports various input formats defining MDPs. Here, we will concentrate on a novel format based on Java programs. In this format, models inherit from a class `Model`, with a few abstract methods. Before we define the exact semantics of the abstract model, we will consider the ACC model as an example.

```
public class ACC extends Model<ACC> {
    // Distance from car in front
    @CVAR(min=0, max=100)
    public double distance;

    // Relative velocity
    @CVAR(min=-14, max=14)
    public double velocity;

    // Distance we desire
    public double desiredDistance = 50;

    // Updates per second
    public int ticks = 10;

    @Override
    public void next(double acceleration, ACC target) {
        double random_acceleration = normal.sample(0, 4.0);
        double nextVelocity =
            velocity + (acceleration + random_acceleration) / ticks;
        double nextDistance = distance -
            (0.5 * velocity + 0.5 * nextVelocity) / ticks;
        target.velocity = nextVelocity;
        target.distance = nextDistance;
    }

    @Override
```

```
        public double[] rewards(double acceleration) {
        double[] rewards = new double[2];
                rewards[0] = -Math.abs(desiredDistance - distance);
                rewards[1] = -acceleration* acceleration;
                return rewards;
        }
}
```

A few features here are noteworthy. Variables that are part of the model are annotated with `@CVAR`. A variable should be part of a model if (1) it influences the probability distribution and (2) it changes over time. In our case, variables `distance` and `velocity` are part of the model, while `desiredDistance` and `ticks` are not part of the model (because they are constant). Method `void next(double acceleration, ACC target)` defines a distribution over the next states, given the current state. We first sample a random acceleration for the other car, with mean zero and standard deviation 4. Next, we calculate the velocity of the next state, based on the current velocity, the random acceleration and the acceleration we got as input. Lastly, based on old velocity and distance and on the new velocity, we calculate the distance of the next state. Note that our framework supports both loops and branching, although they are not present in this example. Additionally, we define the rewards the controller gets for its decisions in method `double[] rewards(double acceleration)`. In the case of `ACC`, it receives a cost (negative reward) depending on how far the current distance is from the desired distance (`rewards[0]`), and a cost depending on how much it accelerates (`rewards[1]`). Note that these two define exactly the trade-off mentioned before. On the one hand, we want to minimize both `rewards[0]` and `rewards[1]`, but applying less acceleration will lead to a greater deviation from our desired distance. On the other hand, being stricter about staying close to the desired distance requires more acceleration.

A class like `ACC` (defined above) by itself defines a continuous Markov Decision Process, i.e., a MDP with infinite sets of states and actions. It describes how a system evolves over time. For example, if we are in a state in which distance is 50 and velocity is -4, then we first have to draw a random sample from a normal distribution to determine how much the car in front of us accelerates or brakes. In our example, we assume that we draw a zero. Therefore, only our own acceleration counts. Since we are already at the desired distance, we can assume that a sensible controller will brake to catch the negative relative velocity. Let us assume that the controller gives us an acceleration of $3m/s^2$. Then the next velocity is going to be $-4 + (3 + 0)/10 = -3.6$. The next distance is defined by the current distance plus the average of old and new distance, i.e. $50 + (-3.6 + -4.0)/2/10 = 49.62$ meters. So now we are in state $(49.62, -3.6)$. Notice how we left the desired distance, but we also decreased our velocity. Presumable, the controller will continue in this way until we have removed all our access velocity, producing more points on the way (e.g., $(49.28, -3.2), (48.89, -2.8), \ldots, (48, 0)$. At this point, the controller will continue braking so that the other car gains distance again, e.g., we will now see points $(48.02, 0.4), (48.08, 0.8) \ldots$. After that (at around distance $49m$) we might see acceleration again such that we reach relative velocity 0 once we reach the desired distance. Note that in this example we assumed that the car in front of the controlled car will maintain its own velocity. In the real model, though, the controller has to cope with the other car's random behavior.

In the trace above, the controller gets rewards at each instant. For example, in the transition between the second two points (from $(49.28, -3.2)$ to $(48.89, -2.8)$), the controller gets two scores: 1. $0.72$ for the deviation from the desired distance, and 2. $-9$ for the applied acceleration. We will discuss later how we combine these instantaneous rewards to rewards over infinite traces and how to combine rewards over traces to rewards for controllers (which define probability measures of infinite sets of infinite traces).

## 4.2 Synthesis

In general, finding perfect controllers for these continuous, probabilistic sequential models is impossible. We still have to deal with them since they are very useful for modelling real world physics. Further, once we decide what action to take in what state, we can use a model like the above for simulation and continuous analysis[3]. In fact, we will use models like `ACC` for Bayesian Model Checking of synthesized controllers later.

### Discretization

To fit such a continuous, infinite state model into our framework, (i.e., finite set of states and actions), we employ sigma point sampling [BH08] and linear interpolation. This is a non-trivial process, but happens transparently for the user. All that is necessary for him is to add an annotated class like in the following.

```
@Discretize(model=ACC.class)
public static class DModel extends ACC {};
```

`DModel` is now a discretized version of `ACC` and describes a MDP as formally defined above. The specific details of discretization (like number of discrete states used) and sampling are configurable. Note that discretized classes like this one are not part of the hidden internals of the framework but are fully accessible to the engineer.

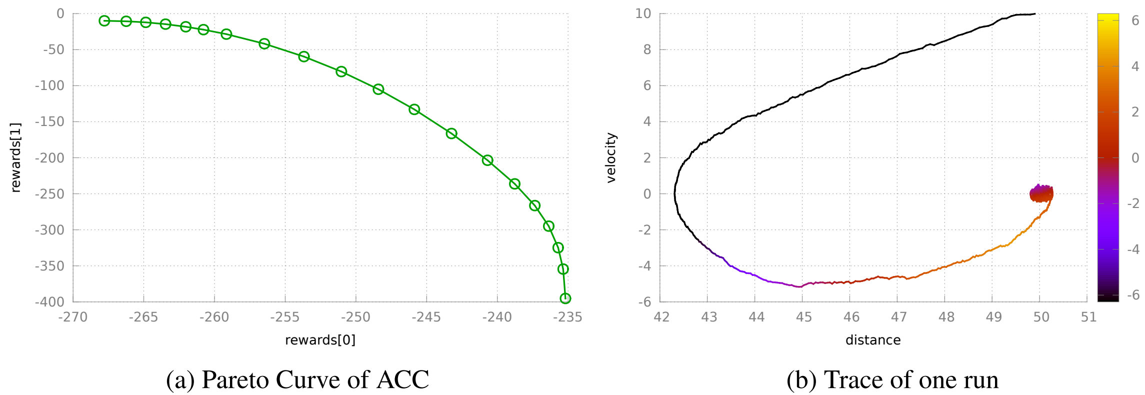
### Algorithms

We support various algorithms for finding optimal controllers. They differ in how they treat the sequence of rewards the evolution of the process presents. Let $r_1, r_2, r_3 \dots$ be a sequence of vectors. Then the aggregated reward is defined in one of the following ways.

- Total Sum: $r_1 + r_2 + r_3 + \dots$
- Discounted: $r_1 + \lambda r_2 + \lambda^2 r_3 \dots$
- Average Payoff: $\lim_{n\to\infty} 1/n \sum_{i=0}^{n} r_i$

When we fix a controller, a model gives us a vector of expected rewards for each state of the model. Together with a distribution over the states, these rewards can be combined into a single vector, which serves as an overall *quality measure*. For two different controllers with rewards $r_1, r_2 \in \mathbb{R}^n$, we say that the controller generating $r_1$ is better than the controller generating $r_2$ if $r_1 > r_2$. In a recent publications [FKP12], the authors showed that all possible quality measures form a convex set for Total Sum and Discounted aggregators. This allows us to effectively approximate the shape of all possible quality measures in form of a Pareto Curve. One example of such a curve is shown in Figure 18a. The curve clearly shows the trade-offs between the two rewards. An engineer might select a region for further refinement, or ask us to analyze all controllers in a certain region of the curve (all of this is supported by our framework). From the extreme right of this curve we can conclude that even small reductions in the importance of the deviation from distance over time leads to massive decreases in applied acceleration in this area. Analogously, the left extreme of the curve shows us that that small reductions in applied acceleration lead to big gains in time spend far away from the desired distance. Using these conclusions, an engineer can pick a quality measure and therefore a controller he desires.

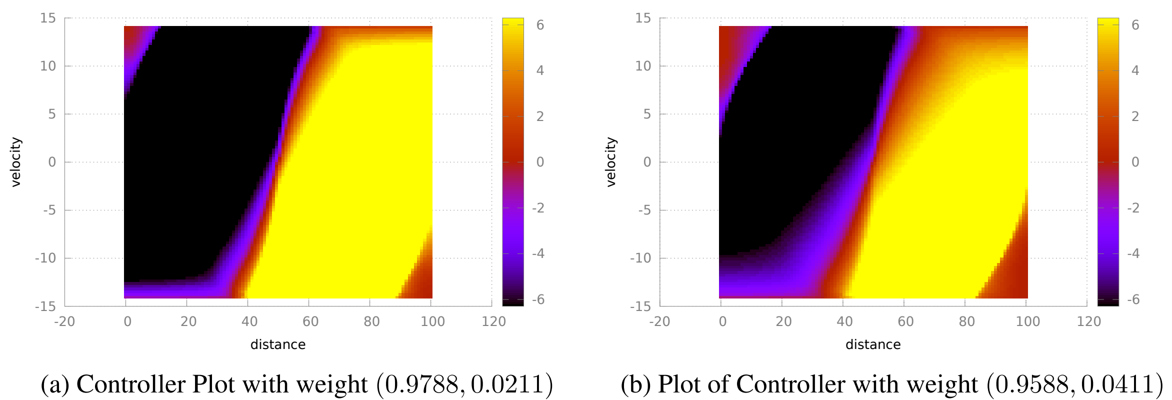
[3] Note that this is indeed the way simulations are usually written




(a) Pareto Curve of ACC

(b) Trace of one run

Figure 18: Pareto curve and trace of controller




(a) Controller Plot with weight $(0.9788, 0.0211)$

(b) Plot of Controller with weight $(0.9588, 0.0411)$

Figure 19: Two controller plots for different weights

We show the actions of a controller generated with weight $(0.9788, 0.0211)$ by our framework in Plot 19a. On the x-axis we present the distance to the car in front, while we present the relative velocity on the y-axis. The color indicates the applied acceleration. For example, where the distance is as desired (50 m), and the relative velocity is 0, no further acceleration is applied. Going through this point is a diagonal going from roughly (35, -15) to (65, 15) where applied acceleration equals zero. In this area, the controller judges the relative velocity just right to reach the desired distance quickly enough. As we move horizontally outwards from this narrow band, the acceleration the controller applies rises sharply. Especially, as either distance or relative velocity decreases, the controller increases the applied acceleration.

Plot 18b shows one trace of the interplay between controller and environment as it happens in the continuous environment (i.e., we run the program defined above as it is). It starts out in position (50,10), i.e., where the distance is as desired but we are closing in too fast. As we follow the trace, we see that the car equipped with an ACC gains on the car in front (as it's velocity is greater than that of the other car). The color of the trace shows the applied braking force in each particular moment. As we can see, the controller brakes the car harshly until he reaches a relative velocity of -3 m/s. At this point it slowly decreases the de-acceleration until a relative velocity of about -5 m/s is reached. It now maintains speed until we reach a distance of about 47 m (i.e., the car is 3 meters too close). Would the controller maintain speed here, then it would overshoot the desired distance. Instead, it gently accelerates the car again until it reaches a relative velocity of 0 and is very close to the desired distance. The "ball" region around the desired distance and relative velocity 0 shows how the controller reacts to the random behavior of the car in front.

In Plot 19b, we present a controller generated with weight $(0.9588, 0.0411)$. In comparison to the weight above, we have decreased the importance of `rewards[0]`, and increased the importance of `rewards[1]`. This decreases the importance of the distance to the other car and increases the importance of not applying too much acceleration. This has the effect of growing the band where relative velocity is judged adequate, and also moving the area of increased acceleration further out.

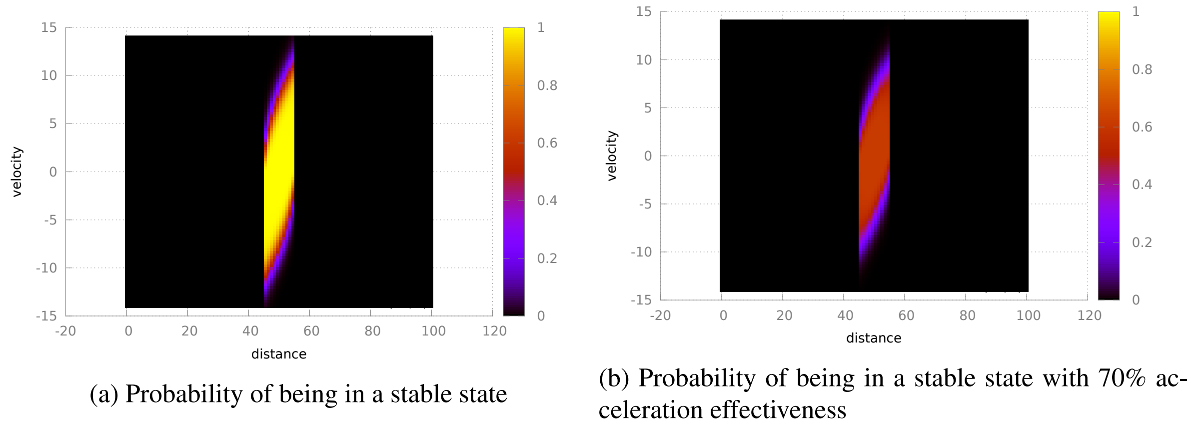
These two examples show that the weight chosen when optimizing a controller can have a strong influence on the one hand, and that choosing weights is not intuitive on the other hand, especially as the number of dimension increases. We therefore consider the easy availability of Pareto Curves an asset of our framework.

## 4.3 Verification

Once a controller has been selected, we can turn to verification and validation. To that end, we support two systems that complement each other: (1) a classical probabilistic model checking algorithm, and (2) a Bayesian probabilistic model checking algorithm.

### Classical Probabilistic Model Checking

We use both systems to judge how the controller behaves if assumptions we made about the environment are not met and how the controller behaves with regard to properties that were not used for its construction. As an example of the latter, we can consider the stability of the system. In control theory, stability is the property of a system to reach a bounded set of states and never leave it. In our case, we define this set as a bound on the deviation of the distance of the two cars from the desired distance. We can easily state a desired bounded set of states via PCTL formula: $P_{=?}[\mathsf{G}(|d-50| < c)]$, where $d$ denotes the distance between the two cars and $c$ is a constant. This formally asks "what is the probability of from now on always (denoted by $\mathsf{G}$) seeing states whose deviation from 50 meters is less than $c$. Our framework takes this formula as input and calculates the probability of being in a stable (i.e., in a state from which only other stable states can be reached) for each state. In Plot 20a we plot




(a) Probability of being in a stable state

(b) Probability of being in a stable state with 70% acceleration effectiveness

Figure 20: Statibilty of the controller

the probability of being in a stable state, where we arbitrarily judge a state stable if $c = 5$. Note, first, that any state with a distance not within 5 meters of the desired distance cannot be stable. Note second, that as the relative velocity becomes more extreme, the probability of a state being stable goes towards zero. At the very extreme ends, the controller is unable to maintain control over the relative velocity in a way that guarantees that the distance will stay within 5 meters of the desired distance. Closer to the area where relative velocity is 0, the probability lies between 0 and 1. The reason that there is no sharp threshold between probability 0 and 1 lies in the random acceleration of the car in front. With a certain probability, the car in front will contribute to moving the distance towards the desired distance (braking where the controller needs to accelerate and vice versa). With a certain probability, the car will work against our controller (accelerating where we need to accelerate, braking where the controllers needs to brake as well).

Judging the probability of reaching a stable state is an additional task. This can be easily done in our framework by checking the controller against formula $P = ?(\mathsf{F}\, P_{=1}[\mathsf{G}(|d - 50| < c)])$. Verbally, this means "what is the probability of reaching a state such that the state is stable almost surely?". As it turns out, the probability is 1 for all states of our model, i.e., under the given assumptions the controller is able to reach and maintain a low deviation from the desired distance to the other car almost surely.

We can now modify certain parameters of the system, and judge its behavior under these modified assumptions. For example, consider very rainy weather, where we assume that acceleration only works at 70% efficiency of what the controller expects[4]. In this case, the probability of a state being stable is only about at most 40% (see Plot 20b)[5].

Lastly, our framework also allows us to easily turn the tables around and choose actions for the car in front. In this new model, the braking force applied by the ACC is determined by a controller we previously generated, and we now synthesize worst-case accelerations for the car in front. This is easily achieved by replace `next` above by the following.

```
public void next(double acceleration2, ACC target) {
    double acceleration = controller.get(this);
    double nextVelocity =
        velocity + (accleration + acceleration2) / ticks;
    double nextDistance = distance −
```

[4]This assumes that we use the same controller in bad weather, and that we cannot compensate

[5]Note that there are techniques for dealing with uncertain parameters(e.g., Robust Markov Decision Processes

```
            (0.5 * velocity + 0.5 * nextVelocity) / ticks;
        target.velocity = nextVelocity;
        target.distance = nextDistance;
    }
```

Now we can apply the very same techniques we used above to compute the worst-case probability of a state being stable.

**Bayesian Probabilistic Model Checking**

As we have noted before, the models described in Java lend themselves directly to continuous state space simulation. We cannot check the PCTL formula above as it is, because it expresses properties over infinite runs. Instead we have to give time-bound formulas. As an example, we consider a formula expressing the property "What is the probability that we reach a state inside 5 meters around the desired distance in 1000 steps (where 1 step is 10 milliseconds long), and stay inside this area for the next 1000 steps." Bayesian probabilistic model checking allows us to make statements like "given the set of samples generated, the probability that this formula is true lies in the interval $[a, b]$ with probability $c$". In this framework, the width of the interval $b - a$ and confidence $c$ are configurable. In our case it turns out, that with 95% confidence the formula holds with probability $[0.98, 1.00]$ from some randomly generated state. We assume that the remaining cases will require longer runs. For comparison, we decreased the efficiency of the applied acceleration to 70%. In this case, we get an interval [0.971297, 0.991297], which shows us that the controller performs well even under adverse conditions.

## 4.4 Conclusion

We believe that our framework is the first time that verification and synthesis are present in a loop in the same tool. It allows engineers to (1) quickly model probabilistic environments for controllers in a language they know, (2) study the trade-offs their model possesses and pick a controller that is to their liking, (3) study the robustness of their controller with respect to environment assumptions, (4) study the performance of the controller in criteria for which the controller was not optimized, (5) allow efficient specification of latter criteria via a formal language, (6) judge the effect of discretization on the same criteria via the simulation engine we contribute. (7) effectively compare the influence discretization resolution has on the controllers.

In addition to what we presented here, our framework is developer-friendly and open and allows quick addition of new synthesis and analysis algorithms. It also allows the easy consumption of new input formats. For example, to judge the correctness of our PCTL model checking algorithm, we imported PRISM [KNP11] models and compared results.

Lastly, the implementation uses parallel algorithms in all performance relevant parts of the system. Speedup is linear in the number of processors, up to the 12 processors we checked.

# 5 Verification of Routing Procedures in Pastry

In [MS12, MS13] and previous deliverables we introduced $\kappa$NCPi, which is an extension of the $\pi$-calculus where processes use network resources (nodes and links) to communicate. One of the key features is that observations of the semantics are routing paths. This allows one to model and prove properties of routing procedures.

Such procedures are important e.g. in the science cloud case study, which uses the peer-to-peer substrate and routing algorithm `Pastry`. The role of `Pastry` in the cloud is that of providing

the network layer (see also Deliverable D7.3) which includes the addressing and message routing functionality on which the more advanced functionality (such as application execution) depend.

In this section we will briefly overview the $\kappa$NCPi model of `Pastry` we presented in [Sam]. More emphasis will be given to formalizing the conditions ensuring that each message eventually reaches its destination in a `Pastry` system. This is informally stated in [RD01], but we need a rigorous formulation so that we can check such conditions in our model.

## 5.1 Routing in `Pastry`.

Each `Pastry` peer is identified by a sequence of digits. Identifiers and keys belong to the same space, and are organized in a virtual ring, ordered in a clockwise fashion. Let $x, y$ two such identifiers: we denote by $shl(x, y)$ the length of the longest prefix shared by $x$ and $y$, and by $d_r(x, y)$ the distance between $x$ and $y$ on the ring, in terms of the number of identifiers between them. We assume that identifiers are totally ordered and that arithmetic operations on them are defined.

The main service provided by `Pastry` is *routing by key*: given a key $k$, `Pastry` delivers the message to the peer which is *responsible for* $k$, i.e. the one whose identifier is closest to $k$ than all other peers. See [RD01] for a detailed description of `Pastry` routing. The following property spells out conditions for routing convergence.

**Condition 5.1** (Routing convergence). *Given a message with target key $k$ and a peer $id$, either $id$ is responsible for $k$ or it should be able to forward the message to $id'$ such that $shl(id', k) > shl(id, k)$ or $d_r(id', k) < d_r(id, k)$.*

## 5.2 The model

Our model is organized into two levels: a *network level*, where we implement `Pastry` networking functionalities, and an *application level*, where we model a simple distributed hash table (DHT) that uses the network-level routing services to accomplish key lookups.

**Network level.** The key idea is modeling identifiers as sites, and the routing table and leaf-set of a peer with identifier $a$ as two collections $\mathcal{L}^a_{RT}$ and $\mathcal{L}^a_{LS}$, respectively, containing links from $a$ to other peers. We use *anonymous* and *typed* links: $a \square b$ is a generic network-level link from $a$ to $b$; $a \boxdot b$ is a link to $b$ in $a$'s routing table; $a \boxminus b$ is a link to $b$ in $a$'s leaf-set.

Our model of a `Pastry` peer implements two functions: handling of node joins and provision of routing services to applications built on top of `Pastry`.

The join procedure aims at setting up routing table and leaf-set of a joining peer. We briefly sketch our implementation. Whenever a peer $a$ receives a join request for a new peer $b$, it always answers with an acknowledgement message via a temporary link $a \square b$. The content of such message depends on how close is $a$ to $b$. If $b$ belongs to the interval spanned by the leaf-set of $a$, then the link $a \boxminus d$ such that $d$ is the closest leaf to $b$ is picked from $\mathcal{L}^a_{LS}$, and we have the following cases: if $d \neq a$, then $a$ forwards the join request to $d$ via $a \boxminus d$, and sends to $b$ an acknowledgement message with the list of peer identifiers in its routing table, namely those identifiers that are target of links in $\mathcal{L}^a_{RT}$; if $d = a$ then the join request does not need to be further forwarded and the response acknowledgement contains the identifiers in $a$'s routing table and leaf-set, respectively; the latter allow $b$ to compute its own leaf-set. If $b$ does not belong to the leaf set, then $a \boxdot c$ such that $shl(b, c) > shl(a, b)$ is selected from $\mathcal{L}^a_{RT}$ and used to forward the request, and an acknowledgement message containing the peer identifiers in $a$'s routing table is sent to $b$.

Whenever $b$ receives a join acknowledgement, it updates its leaf-set and routing table with fresh links to the peer identifiers carried by the message. After receiving the last acknowledgement, $b$ sends

a notification message to all the peers in its routing data structures. These peers will subsequently update their routing data with new links to $b$.

A Pastry system with $n$ peers $a_1, \ldots, a_n$ is modeled as a process of the form

$$\mathtt{Peer}(a_1, \mathcal{L}_{LS}^{a_1}, \mathcal{L}_{RT}^{a_1}) \,|\, \mathtt{Peer}(a_2, \mathcal{L}_{LS}^{a_2}, \mathcal{L}_{RT}^{a_2}) \,|\, \ldots \,|\, \mathtt{Peer}(a_n, \mathcal{L}_{LS}^{a_n}, \mathcal{L}_{RT}^{a_n}) \ .$$

For such systems, we have the following result about routing convergence.

**Theorem 5.2.** *Let $a_{n+1}$ be the identifier of a peer that intends to join the system. Then, after the join procedure for $a_{n+1}$ has ended, the following property holds: for each key $k$ and each $a_i$, either $a_i$ is responsible for $k$ or there a link from $a_i$ to some $b$ in $\mathcal{L}_{RT}^{a_i} \cup \mathcal{L}_{LS}^{a_i}$ such that $b$ is closer to $k$ than $a_i$, i.e. $a_i, b$ and $k$ satisfy condition 5.1 with $id = a_i$ and $id' = b$.*

Our peer model also implements provision of routing services to applications: it picks any link in $\mathcal{L}_{LS}^{a} \cup \mathcal{L}_{RT}^{a}$ and makes it available to the application level via a special primitive we included in the language. The selection of which link to use will be made at application-level.

**Application level.** Now we show how routing behavior for a simple Distributed Hash Table can be modeled. We define a new transition system $\rightsquigarrow$ where observations are routing paths taken by DHT lookups. We introduce the following types of links: $a \rhd b$ is a generic application-level link from $a$ to $b$; $a \trianglerighteq k$ is a link indicating that $a$ is responsible for the key $k$ in the DHT.

The semantics has a new kind of observation: $a; W; b\{k\}$ is a composite routing service that can only be used by routing paths with target key $k$. Inference rules are of two kinds: those translating network-level routing services to application level ones and those synchronizing processes by concatenating their paths. The former implement Pastry routing mechanism at the SOS level by selecting the correct service to be used by a peer when forwarding towards a given key. For instance we have

$$\frac{p \xRightarrow{a;a\boxminus b\uparrow;b} p' \quad p \not\xRightarrow{a;a\boxminus d\uparrow;d} \quad p \xRightarrow{a;a\boxminus c_1\uparrow;c_1} p_1 \quad p \xRightarrow{a;a\boxminus c_2\uparrow;c_2} p_2}{p \overset{a;a\rhd b;b\{k\}}{\rightsquigarrow} p'} \quad \begin{array}{l} c_1 \prec k \prec c_2 \\ d_r(k,d) < d_r(k,b) \end{array}$$

which says that $a$ is allowed to provide a link to some $b$ in its leaf-set only if the target key $k$ belongs to $a$'s leaf-set range and $k$ is closer to $b$ than to any other leaf of $a$. As for the other kind of rules, the synchronizations mechanism is still based on name matching, like in the ordinary $\kappa$NCPi semantics, but now also involves the target key: only paths that meet at the same site and have the same target key can be concatenated.

We can model a Distributed Hash Table over a Pastry system with peers $a_1, \ldots, a_n$ as follows. Suppose the DHT has $m$ key-value pairs $\langle k_i, v_i \rangle$, and let $a_{k_i}$ be the identifier of the peer responsible for $k_i$, i.e. the closest to $k_i$ among $a_1, \ldots, a_n$. Then we have

$$\mathtt{DHT} \overset{\text{def}}{=} \mathtt{Peer}(a_1) \,|\, \ldots \,|\, \mathtt{Peer}(a_n) \,|\, \mathtt{H} \qquad \mathtt{H} \overset{\text{def}}{=} \mathtt{Entry}(k_1, v_1, a_{k_1}) \,|\, \ldots \,|\, \mathtt{Entry}(k_m, v_m, a_{k_m})$$

$$\mathtt{Entry}(k, v, a) \overset{\text{def}}{=} a \trianglerighteq k \,|\, k(b).\overline{a}bv.\mathtt{Entry}(k, v, a)$$

Here H represents the DHT content as the parallel composition of processes that handle the table's entries. The idea is implementing a DHT lookup request for a key $k$ as a message with destination $k$, carrying the identifier $b$ of the sender. Upon receiving this message, the handler for $\langle k, v \rangle$ replies to $b$ with a message containing $v$. See [Sam, §6.4] for an example.

We have the following results about routing convergence. Theorem 5.3 states that the system is always able to provide a link from any peer $a$ towards a given key $k$, and the target of this link is a peer closer to $k$ than $a$, according to 5.1. As a consequence, there is always a path able to route a lookup

request from $a$ to the peer $a_k$ responsible for $k$ (see Corollary 5.2); this path is an input path (i.e. has $\bullet$ on the right), as it represents the fact that $a_k$ is waiting for lookup requests from processes in the execution context.

**Theorem 5.3.** *For every peer $a$ and key $k$ there is a transition* $\mathtt{DHT} \xrightarrow{a;a\triangleright b;b\{k\}} \mathtt{DHT}'$, *where either $b = k$ or $a, b, k$ satisfy condition 5.1.*

**Corollary.** *Let $k$ be a key in the DHT and $a_k$ the peer responsible for it. Then, for every peer $a$, there exists a transition* $\mathtt{DHT} \xrightarrow{aa_k a;a\triangleright a';\ldots;a_k\triangleright k;\bullet} \mathtt{DHT}'$.

# 6 Stable Availability under Denial-of-Service Attacks through Formal Patterns

On December 8, 2010 at 07:53 AM EDT, MasterCard issued a statement that "MasterCard is experiencing heavy traffic on its external corporate website [...]. There is no impact whatsoever on our cardholders ability to use their cards for secure transactions" [Mas]. In fact, by that time, a distributed denial-of-service attack (DoS) brought the website down and made their web presence unavailable for most customers for several hours.

Availability is an important security property for Internet services and a key ingredient of most service level agreements. It can be compromised by distributed denial-of-service (DoS) attacks. DoS defense mechanisms help maintaining availability; nevertheless even when equipped with defense mechanisms, systems will typically show performance degradation. Thus, one of the goals of security measures is to achieve stable availability, which means that with very high probability service quality remains very close to a static constant quantity, regardless how bad the DoS attack gets.

Cloud computing offers the possibility of dynamic resource allocation and thus can be used to leverage stable availability when combined with DoS defense mechanisms.

**Contributions.** In our work we propose a formal pattern-based approach to study defense mechanisms against DoS attacks. We enhance pattern descriptions with formal models that allow the designer to give guarantees on the behavior of the proposed solution. The underlying executable specification formalism we use is the rewriting logic language Maude and its real-time and probabilistic extensions.

We introduce the notion of stable availability, which means that with very high probability service quality remains very close to a threshold, regardless of how bad the DoS attack can get. We use formal patterns which, in addition to "normal" patterns, come with formal guarantees and enable automated pattern composition, often resulting in semi-automatic construction of new models with improved properties. We use this pattern-based approach to study defense mechanism against DoS attacks in a model-based setting.

We present two formal patterns which can serve as defenses against DoS attacks:

- the Adaptive Selective Verification (ASV) pattern, which enhances a communication protocol with a defense mechanism, and
- the Server Replicator (SR) pattern, which provisions additional resources on demand.

However, ASV achieves availability without stability, and SR cannot achieve stable availability at a reasonable cost. We thus adopt the ASV$^+$SR protocol, which combines the Server Replicator with the ASV Wrapper to achieve protection against DDoS attacks in two dimensions of adaptation: (i)

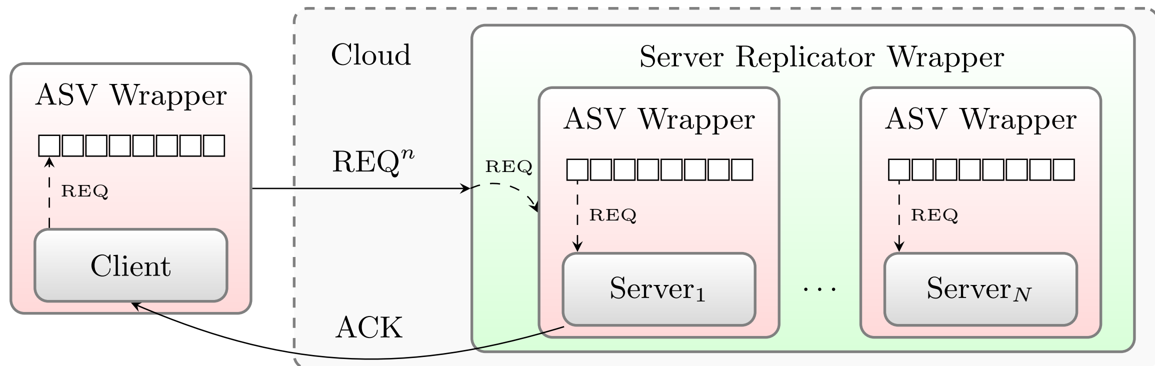



Figure 21: Application of the ASV$^+$SR meta-object composition on a Cloud-based client-server request-response service

adapting to increasingly more severe DoS attacks using the ASV mechanism; and (ii) adapting to the increasing need for server performance using the SR mechanism.

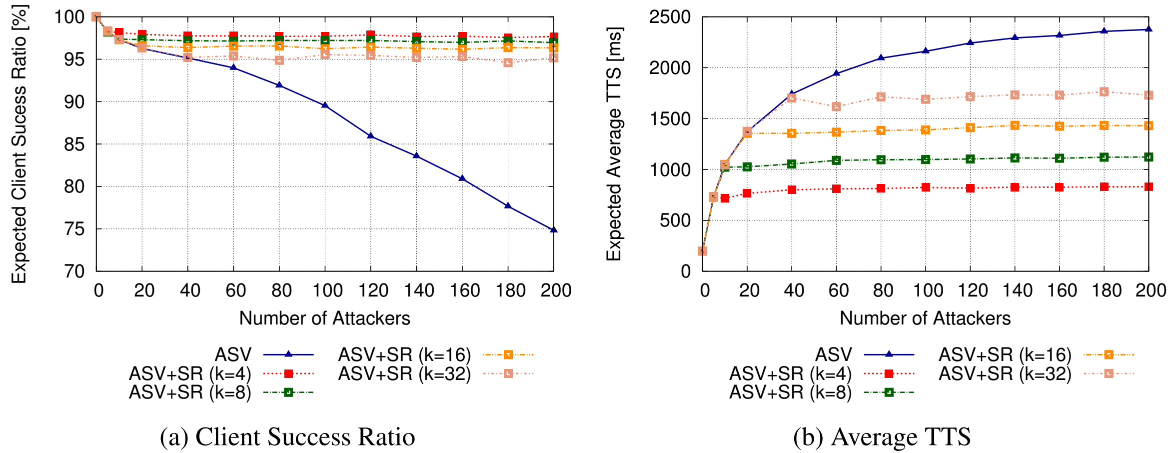
An overview of a Cloud-based service setup that uses the ASV$^+$SR protocol is shown in Figure 21. Clients and servers still adapt to a possible attack by exponentially increasing the number of requests on the client-side and by collecting a random sample of incoming requests on the server-side. However, the entry-point for all requests on the server-side is no longer a single server but the Server Replicator meta-object. The meta-object wraps around server instances which are themselves wrapped by the server-side ASV Wrapper. In the ASV$^+$SR protocol, the maximum load per server is equal to the product of its time out window size and the servers mean processing rate ($T * S$).

For the replication metric, an additional parameter k, namely, the server overloading factor, is defined. The metric says that a new server is spawned by the Server Replicator, if the servers are overloaded by the overloading factor times their maximum load, e.g., for a factor of $k = 4$ and a maximum load of 10 $REQs$ per second per server, the Server Replicator spawns a new server if the wrapped servers have a load average that is greater than 40 $REQs$ per second per server. Thus, the factor k defines by how much an ASV$^+$SR protected system uses the selection mechanism of the server-side ASV Wrapper. An overloading factor of $k = 1$ means that the ASV protocol is nearly unused, an overloading factor of $k = 1$ means that only the ASV protocol is used, because additional servers are never provisioned by the Server Replicator. We therefore propose an overloading factor $k$ with $1 < k < \infty$ to be used with the ASV$^+$SR protocol.

We use the Maude-based specification of the ASV$^+$SR meta-object pattern with a client-server system to perform parallelized statistical quantitative model checking on 20 to 40 cluster nodes using PVESTA. The expected values of the following QUATEX path expressions were computed with a 99% confidence interval of size at most 0.01:

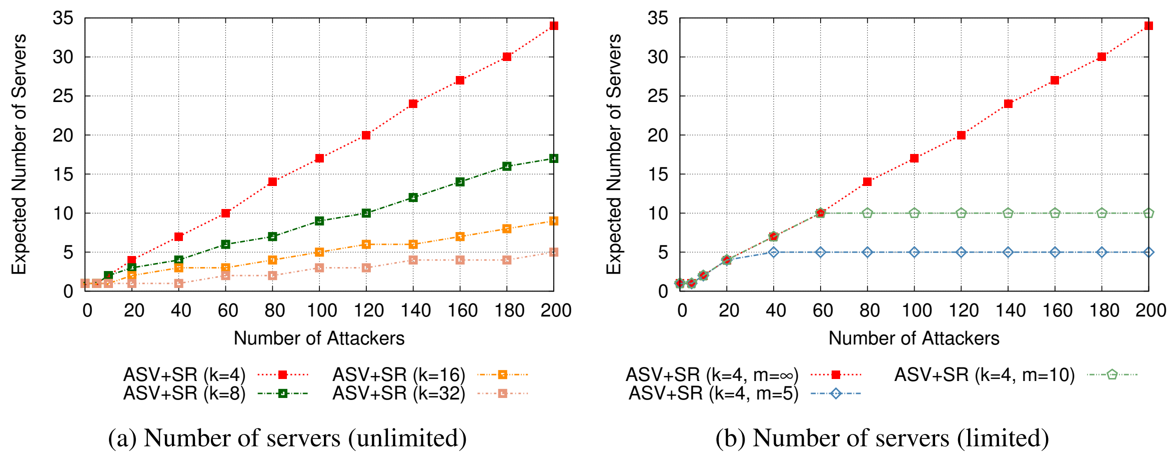
We have checked the following three properties:

- *Client success ratio* The client success ratio defines the ratio of clients that receive an acknowledgement from the server
- *Average Time-to-Service (TTS)* The average TTS is the average time it takes for a successful client to receive an acknowledgement from the server
- *Number of servers* The number of servers represents the number of ASV servers that are spawned by the SR meta-object




(a) Client Success Ratio (b) Average TTS

Figure 22: Performance of the ASV$^+$SR protocol with a varying load factor $k$ and no resource bounds




(a) Number of servers (unlimited) (b) Number of servers (limited)

Figure 23: Expected number of servers using the ASV$^+$SR protocol.

The properties are checked for a varying number of attackers (1 to 200). Each attacker issues 400 fake requests per second. It is of note that 1.5 attackers already overwhelm a single server. The values of the ASV and attack parameters correspond to the values chosen in [AMG09, KVF$^+$08]. Additionally, an initial generation delay of 0.05 seconds is introduced and the duration of a simulation is set to 30 seconds.

In the following, we will consider two general cases in which the SR can provision: (i) an unlimited number of servers, and (ii) servers up to a limit $m$ of 5 or 10 servers, because, due to economical and physical restrictions, resources are limited. The results in (i) will indicate how many servers are needed to provide stable service guarantees, while the results in (ii) will indicate what service guarantees can still be given with limited resources.

**Unlimited resources.** Fig. 22 shows the model checking results for a varying overloading factor $k$ with no resource limits. As indicated by Fig. 22a, ASV$^+$SR can sustain the expected client success ratio at a certain percentage. Even for an overloading factor of $k = 32$, a success ratio around 95% can be achieved. Compared to an overloading factor of $k = 4$, a 7-fold decrease in provisioned servers is observed (Fig. 23a), achieving a stable success ratio of only around 3% less. Fig. 22b shows that the same is true for the average TTS. ASV$^+$SR outperforms the ASV protocol, and furthermore achieves

stable availability, for all performance indicators. However, this comes at the cost of provisioning new servers. Fig. 23a shows how many servers are provisioned. The results indicate that the factor $k$ defines a trade-off between the cost and the performance of stable availability. SR by itself ($k = 1$) with unlimited resources (not shown in the figures) would provide stable availability at a level as if no attack has happened, but would provision 134 servers for 200 attackers. Note that fluctuations in the results, e.g., the average TTS in case of 60 attackers being lower than the average TTS in case of 40 attackers, are due to the provisioning of a discrete number of servers.

**Limited resources**. In the case of limited resources (i.e. maximum fixed number of servers), the protocol behaves just as in the case of unlimited resources up to the point where more servers than the limit would be needed to keep the success ratio stable. After that point, the protocol behaves like the original ASV protocol (but with the equivalent of a more powerful server) and the success ratio decreases.

For further details on this work we refer the reader to our publications [WEMM12, EMA+12].

# 7 Access Control, Resource Usage, and Adaptation Policies for a Cloud Scenario

In this section, we briefly present a development methodology for policy-based systems and its application to a Cloud IaaS scenario. Our methodology is based on FACPL [MMPT13a], a policy language capable of dealing with different systems' aspects through a user-friendly, uniform, and comprehensive approach. Indeed, FACPL can express access control policies as well as policies dealing with other systems' aspects, as e.g. resource usage and adaptation.

FACPL intentionally takes inspiration from XACML [OAS12], the OASIS standard language for defining access control policies, but is much simpler and usable. Differently from XACML, FACPL has a compact and intuitive syntax and is endowed with a formal semantics based on solid mathematical foundations, which make it easy to learn and, most of all, paves the way to reasoning about policies. Moreover, in FACPL policies can be written at a higher abstraction level than XACML.

The development and the enforcement of FACPL policies is supported by practical software tools: a powerful Eclipse-based development environment and a Java library supporting the policy evaluation process. The policy designer can use the dedicated environment for writing the desired policies in FACPL syntax, by taking advantage of the supporting features provided by the tool. Then, according to the rules defining the language's semantics, the tool automatically produces a set of Java classes implementing the FACPL policies. The generated policy code can be integrated as a module into the enclosing application and can be used to compute a policy decision by executing it with the request code passed as parameter.

## 7.1 A Cloud IaaS Scenario

We consider here a scenario from the Cloud computing domain, in which a small-size IaaS provider offers to customers a range of pre-configured *virtual machines* (VMs), providing different amounts of dedicated computing capacity in order to meet different computing needs. Each type of VM features a specific *Service Level Agreement* (SLA) that the provider commits to guarantee. Thus, the allocation of the right amount of resources needed to instantiate new VMs (while respecting committed SLAs) is a key aspect of the considered IaaS provider. As is common for Cloud systems, virtualisation is accomplished using an *hypervisor*, i.e., a software entity managing the execution of VMs.

For the sake of simplicity, the considered IaaS provider relies only on two hypervisors (i.e., `HYPER_1` and `HYPER_2`) running on top of two physical machines. The provider offers strongly defined types of VMs, like most of popular IaaS providers (consider, e.g., the instance types *M1 Small*

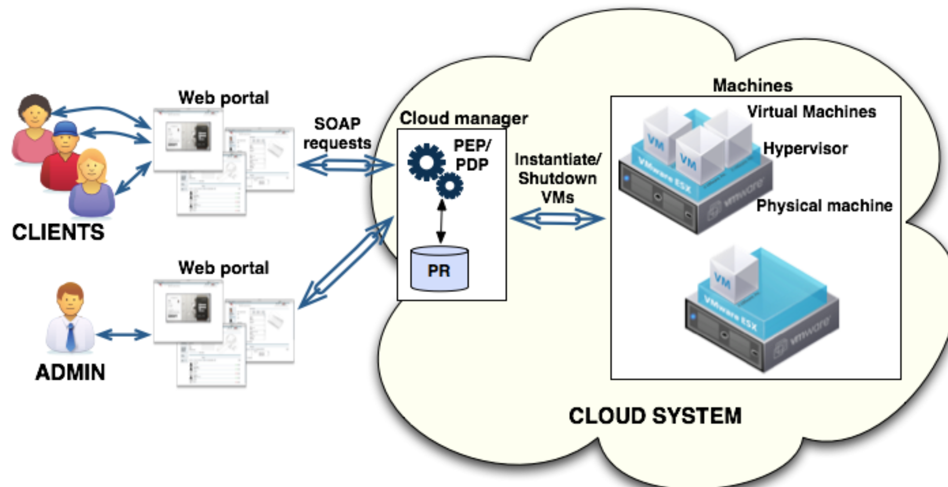



Figure 24: IaaS provider scenario

and *M1 Medium* provided by Amazon EC2). Two types of VMs, namely `TYPE_1` and `TYPE_2`, are in the provider's service portfolio. Each type of VM has an associated SLA describing the hardware resources needed to instantiate the VM (e.g., CPU performance, size of memory and storage) by means of an aggregated measure: `TYPE_1` requires the allocation of one *unit of resources*, while `TYPE_2` requires two units.

The two types of VMs have different guarantees when the system is highly loaded. Specifically, if the system does not have enough resources for allocating a new `TYPE_2` VM, an appropriate number of `TYPE_1` VMs already instantiated will be frozen and moved to a queue of suspended VMs. This queue is periodically checked with the aim of trying to reactivate suspended VMs. When a VM is frozen, according to the *Insurance* [Ver04] SLA approach for resource provisioning in Cloud computing systems, the VM's owner will receive a credit that can be used, e.g., for activating new VMs or for paying computational time.

## 7.2 Policy-based Implementation

A graphical representation of the data-flow in our implementation of the scenario is shown in Figure 24. Clients interacts with the Cloud system via a Web portal that, following a multi-tenancy architecture, sends VM instantiation requests to the *Cloud manager* through SOAP messages. This means that the manager exposes its functionalities to users by means of a Web service. Then, the manager evaluates the received requests with respect to a set of policies defining the logic of the system. In particular, such policies specify the credentials the clients have to provide in order to access the service (*access control policies*), the resource allocation strategy (*resource-usage policies*), and the actions to be performed to fulfill the requests by also taking into account the current system state, which include the system re-configuration actions in case of high load (*adaptation policies*). It is worth noticing that all policies are written by using the same policy language, FACPL, and are enforced by means of the same software tool. By means of a similar workflow, clients can request the shutdown of VMs, which involves the release of the allocated resources.

The administrator of the Cloud system can access a dedicated panel for managing the governing policies. Indeed, he can change at run-time the current policies with other ones, obtaining in this way a fully configurable and adaptable system. The core of the Cloud manager is the *Policy Enforcement Point* (PEP), which evaluates client requests according to the available policies in the *Policy Repository* (PR) and the environmental information about the Cloud system. The sub-component *Policy Decision Point* (PDP) has the duty of calculating if a request can be granted or rejected, and determining the actions needed to enforce the decisions (called *obligations* in FACPL jargon), such as creation, freez-

ing and shutdown of VMs. The enforcing is executed by the PEP by sending to the hypervisors the commands corresponding to the obtained actions. Notably, policies are independent from the specific kind of hypervisors installed on the system, such as XEN or Linux-KVM, i.e., the actions returned by the PDP are converted by the PEP into the appropriate commands accepted by the used hypervisors. Thus, in principle, the policy engine we have developed could be integrated with any IaaS system provided that the adequate action translation is also defined.

We have developed two different approaches for managing, instantiating and releasing requests. The first one concentrates the workload on hypervisor `HYPER_1`, while hypervisor `HYPER_2` is only used when the primary one is fully loaded. Thus, by keeping the secondary hypervisor in stand-by mode until its use becomes necessary, energy can be saved. The second approach, instead, balances the workload between the two hypervisors.

An excerpts of the *energy saving* policies is presented below (we refer the interested reader to [MMPT13b] for a complete account). This specification defines a PEP using the enforcing algorithm `deny-biased`[6] and a PDP using the combining algorithm `permit-overrides`[7], and includes a policy set, for supervising VMs instantiation requests (specifying action `CREATE`), and a policy, for supervising release requests (specifying action `RELEASE`). Such policies are included through a cross name reference, which simplifies code organization.

```
{
 pep: deny-biased;
 pdp: permit-overrides
      include Create_Policies
      include Release_Policies
}
```

The policy set `Create_Policies` uses the combining algorithm `permit-overrides` and specifies a policy for each type of VM, namely `SLA_Type1` and `SLA_Type2`, and a *target* determining the requests to which the policy set applies, i.e. all requests having attribute `action/action-id` set to `CREATE`.

```
PolicySet Create_Policies { permit-overrides
   target:
     equal("CREATE",action/action-id)
   policies:
     Policy SLA_Type1 < ... >
     Policy SLA_Type2 < ... >
}
```

The enclosed policies achieves the prioritized choice between the two hypervisors by specifying the combining algorithm `deny-unless-permit` and by relying on the rules order. As an example, we report below the policy managing the instantiation of `TYPE_1` VMs. The policy's target indicates that instantiation of `TYPE_1` VMs can be required by clients having `P_1` or `P_2` as profile. The policy's combining algorithm evaluates the enclosed rules according to the order they occur in the policy; then, if one of them evaluates to `permit`, the evaluation terminates. Rule `hyper_1` evaluates to `permit` only if the hypervisor `HYPER_1` has at least one unit of available resources and, in this case, returns an obligation requiring the PEP to create a VM in this hypervisor. Rule `hyper_2`, governing VMs creation on `HYPER_2`, is similar. If no rule evaluates to `permit`, then the combining algorithm returns `deny` and, hence, the policy's (optional) obligation will be executed by the PEP to notify the Cloud administrator that there are not enough resources in the system to instantiate a new `TYPE_1` VM. In

[6] The algorithm `deny-biased` states: if the PDP decision is `permit` and all obligations are successfully discharged, then the PEP grants access, otherwise it forbids access.

[7] The algorithm `permit-overrides` states: if any policy among the considered ones evaluates to `permit`, then the decision is `permit`; otherwise, if all policies are found to be `not-applicable`, then the decision is `not-applicable`; in the remaining cases, the decision is `deny` or `indeterminate` according to specific error situations (see [MMPT13a]).

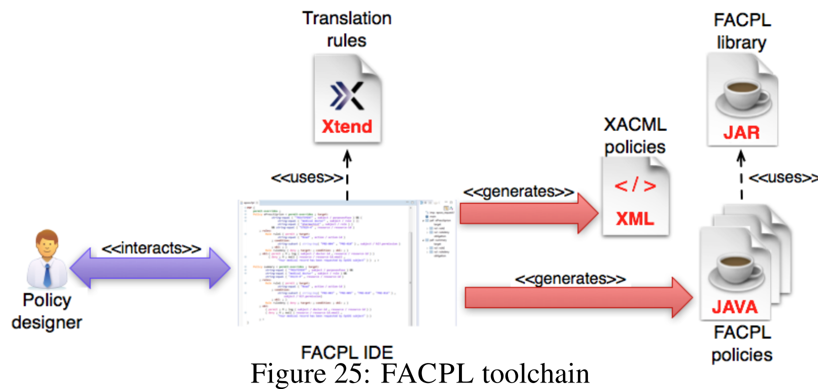



Figure 25: FACPL toolchain

this way, the administrator can decide to upgrade the system by adding new resources (e.g., a new physical machine).

```
Policy SLA_Type1 < deny-unless-permit
  target:
    (equal("P_1", subject/profile-id)||equal("P_2", subject/profile-id))
    & & equal("TYPE_1", resource/vm-type)
  rules:
    Rule hyper_1 (permit
      target:
        less-than-or-equal(1, system/hyper1.availableResources)
      obl:
        [permit M create("HYPER_1", system/vm-id, "TYPE_1")]
    )
    Rule hyper_2 ( ... )
  obl:
    [deny O warning("Not enough available resources for TYPE_1 VMs")]
>
```

The policies for the *load balancing* approach are the same as before except that a condition on the hypervisors' load is added to each instantiation rule. This condition permits applying a rule for a certain hypervisor only if its amount of available resources is greater than or equal to the amount of available resources of the other hypervisor. For example, the rule `hyper_1` is extended as follows:

```
Rule hyper_1 ( permit
  target: ...
  condition: less-than-or-equal(system/hyper2.availableResources,
                                   system/hyper1.availableResources)
  obl: ... )
```

## 7.3 Supporting tools

We have seen so far how the FACPL language can be used to define policies for the considered Cloud scenario. We conclude by briefly describing the software tools we have used to develop and enforce FACPL policies (we refer to Deliverable D6.3 for a more complete account of these tools).

Figure 25 shows the toolchain supporting the use of FACPL. The FACPL Integrated Development Environment (IDE) allows the policy designer to specify the system policies in FACPL. In addition to policies, the IDE permits also specifying user requests in order to test and validate the policies. The specification task is facilitated both by the high abstraction level of FACPL and by the graphical interface provided by our IDE. By exploiting some translation rules, written using the Xtend language, which provides facilities for defining code generators, the IDE generates the corresponding low-level policies both in Java and in XML. The latter format obeys the XACML 3.0 syntax and can be used to connect our toolchain to external XACML tools (as, e.g., the test cases generator X-CREATE [BDLM12]). The former format relies on a Java library specifically designed for compile-

and run-time supporting FACPL code. Once these Java classes are compiled, they can be used by the enclosing main application (i.e., the Cloud manager in our scenario) for evaluating many requests, simply by means of a method invocation. Whenever new policies are introduced, new Java classes will be generated and compiled.

As an example of Java code generation, we report below an excerpt of the code corresponding to the policy `SLA_Type1`:

```
public class Policy_SLA_Type1 extends Policy {
    public Policy_SLA_Type1() {
        addId("SLA_Type1");
        addCombiningAlg(it.unifi.facpl.lib.algorithm.DenyUnlessPermit.class);
        addTarget(new TargetTree(Connector.AND,
           new TargetTree(...), new TargetTree(...)));
        addRule(new hyper_1());
        addRule(new hyper_2());
        addObligation(new ObligationExpression("warning",Effect.DENY,
            TypeObl.O,"Not enough available resources for TYPE_1 VMs"));
    }
    private class hyper_1 extends Rule {
        hyper_1 (){
          addId("hyper_1");
          addEffect(Effect.PERMIT);
          addTarget( ... );
          addConditionExpression(null);
          addObligation(new ObligationExpression ("create",Effect.PERMIT,
            TypeObl.M,"HYPER_1",new StructName("system","vm-id"), "TYPE_1"));
        }
    }
    private class hyper_2 extends Rule { ... }
}
```

Policy evaluation is coordinated by the class implementing the combining algorithm (i.e., `DenyUnlessPermit.class`). The expression corresponding to the policy target is structured as nested expressions organized according to the structure of the original FACPL target. Since rules are only used inside their enclosing policy, for each of them the policy class contains an inner class. In the code above, these are the classes `hyper_1` and `hyper_2`.

In order to experiment with the Java code generated from the complete FACPL specification of the Cloud manager behavior, we have integrated such code with a web application providing both a front-end for the administrator, from where he can manage the policies governing the hypervisors, and a front-end for the clients, from where they can submit requests for the creation or the shutdown of VMs. The server-side implementation is a Tomcat server that, by integrating FACPL and Xtext libraries, is able to accept FACPL polices, to parse and compile these policies, and finally to enforce the corresponding decisions for adapting hypervisors' state to client requests.

# 8 Conclusion

We have presented the application of the verification and validation phase of the ASCENS development lifecycle to case studies of the project. In principle, the first step of this phase is to verify correctness of the designed system based on a functional model. An instance of this step is given by compositional verification applied to a robotics scenario, which permits us to prove that a system of robots cooperating is real-time is safe. We also demonstrated that the networking algorithms proposed for the Science Cloud case study route messages so that they eventually reach their destination, which is a minimal requirement stating that they are intrinsically correct. One can be also interested in the delays encountered to send the messages, but this was not part of the proposed study.

The second step of the validation phase is to check that the system reaches the desired level of performance. For this we need finer models to quantify the amount of used resources such as time,

energy, etc. The proposed models are stochastic allowing not only to represent ranges of behavior in the same model, but also to evaluate their likelihood. We used statistical model-checking to evaluate average case behavior for swarm robotics applications, and to provide probability-based guarantees of system performance. This helped us in finding the most suited algorithms for implementing robots behavior. Statistical model-checking was also used to demonstrate that the pattern $ASV^{+}SR$ for the cloud case study performs better than patterns ASV or SR considered alone, even in presence of a large number of attackers. We also generated controllers for adaptive cruise control which can be included in the e-Mobility scenario, using control synthesis and verification techniques integrated in a single framework.

For the next year, we plan to complete stochastic models we built for the robotics case study to apply statistical model-checking on the entire scenario. We also want to apply (stochastic) abstractions to these models in order to speed up the execution of the tools. Such abstractions must preserve properties of the system we want to check, which will have to be proven.

We plan to improve the current version of MISSCEL by the integration of other reasoners. Another way to improve tools integration in the ASCENS project is to extend MISSCEL to deal with existing stochastic extensions of SCEL. We are also interested in applying MISSCEL to other scenarios.

For control synthesis we mainly want to improve existing tools by speeding up synthesis as well as verification. The idea is to integrate state of the art value iteration acceleration techniques, which have been presented and experimentally tested in the community of formal verification, but which no tool currently implements.

We plan to contribute to a joint paper about Science Cloud, where the `Pastry` model is used to address part of the Science Cloud Patform design. Moreover, we are interested in developing a $\kappa$NCPi model of a cloud system. This could be a further level on top of the `Pastry` model.

We plan to continue the validation of FACPL and its tools by applying them to scenarios from the other two ASCENS case studies. Moreover, we intend to integrate the FACPL evaluation environment within the jRESP runtime environment, thus enabling a full-evaluation of the policy layer when programming ensembles using SCEL. Another research line we intend to pursue is the development of methods and techniques for analyzing FACPL policies. In particular, they will be first theoretically defined and, then, integrated in our software tools in order to achieve a complete framework for developing trustworthy policies.

# References


[AM11] Musab AlTurki and José Meseguer. Pvesta: A parallel statistical model checking and quantitative analysis tool. In *CALCO*, pages 386–392, 2011.

[AMG09] M. AlTurki, J. Meseguer, and C. Gunter. Probabilistic modeling and analysis of DoS protection for the ASV protocol. *ENTCS*, 234:3–18, 2009.

[AMS06] Gul A. Agha, José Meseguer, and Koushik Sen. PMaude: Rewrite-based specification language for probabilistic object systems. In Antonio Cerone and Herbert Wiklicky, editors, *QAPL 2005*, volume 153(2) of *ENTCS*, pages 213–239. Elsevier, 2006.

[BBB+11] Ananda Basu, Saddek Bensalem, Marius Bozga, Paraskevas Bourgos, and Joseph Sifakis. Rigorous system design: The bip approach. In Zdenek Kotásek, Jan Bouda, Ivana Cerná, Lukás Sekanina, Tomás Vojnar, and David Antos, editors, *MEMICS*, volume 7119 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2011.

[BBD+12] Saddek Bensalem, Marius Bozga, Benoît Delahaye, Cyrille Jégourel, Axel Legay, and Ayoub Nouri. Statistical model checking qos properties of systems with sbip. In Tiziana

Margaria and Bernhard Steffen, editors, *ISoLA (1)*, volume 7609 of *Lecture Notes in Computer Science*, pages 327–341. Springer, 2012.

[BBNS10] Saddek Bensalem, Marius Bozga, Thanh-Hung Nguyen, and Joseph Sifakis. Compositional verification for component-based systems and application. *IET Software*, 4(3):181–193, 2010.

[BCG+12a] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch-Lafuente, and Andrea Vandin. A conceptual framework for adaptation. In Juan de Lara and Andrea Zisman, editors, *FASE*, volume 7212 of *Lecture Notes in Computer Science*, pages 240–254. Springer, 2012.

[BCG+12b] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch-Lafuente, and Andrea Vandin. Modelling and analyzing adaptive self-assembly strategies with Maude. In Francisco Durán, editor, *WRLA 2012*, volume 7571 of *LNCS*, pages 118–138. Springer, 2012.

[BCG+12c] Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch-Lafuente, and Andrea Vandin. Modelling and analyzing adaptive self-assembly strategies with Maude. *Submitted to Science of Computer Programming*, 2012.

[BDLM12] Antonia Bertolino, Said Daoudagh, Francesca Lonetti, and Eda Marchetti. The X-CREATE Framework - A Comparison of XACML Policy Testing Strategies. In *WEBIST*, pages 155–160. SciTePress, 2012.

[BDVW] Lenz Belzner, Rocco De Nicola, Andea Vandin, and Martin Wirsing. Reasoning (on) Service Component Ensembles in Rewriting Logic. To appear in the proceedings of SAS 2014, Springer LNCS Festschrift.

[Bel13] Lenz Belzner. Action programming in rewriting logic (technical communication). *Theory and Practice of Logic Programming, On-line Supplement*, 2013.

[BH08] L. F. Bertuccelli and J. P. How. Robust Markov decision processes using sigma point sampling. In *American Control Conference (ACC)*, pages 5003–5008, 11-13 June 2008.

[BJMS12] Marius Bozga, Mohamad Jaber, Nikolaos Maris, and Joseph Sifakis. Modeling dynamic architectures using dy-bip. In Thomas Gschwind, Flavio De Paoli, Volker Gruhn, and Matthias Book, editors, *Software Composition*, volume 7306 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2012.

[BLM+10] Michael Bonani, Valentin Longchamp, Stphane Magnenat, Philippe Rtornaz, Daniel Burnier, Gilles Roulet, Florian Vaussard, Hannes Bleuler, and Francesco Mondada. The MarXbot, a Miniature Mobile Robot Opening new Perspectives for the Collective-robotic Research. In *International Conference on Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ*, IEEE International Conference on Intelligent Robots and Systems, pages 4187–4193. IEEE Press, 2010.

[CDE+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. *All About Maude*, volume 4350 of *LNCS*. Springer, 2007.

[EMA+12] Jonas Eckhardt, Tobias Mhlbauer, Musab AlTurki, Jos Meseguer, and Martin Wirsing. Stable availability under denial of service attacks through formal patterns. In *15th International Conference on Fundamentals of Software Engineering (FASE'12)*, LNCS. Springer, 2012.

[FKP12] V. Forejt, M. Kwiatkowska, and D. Parker. Pareto curves for probabilistic model checking. In S. Chakraborty and M. Mukund, editors, *Proc. 10th International Symposium on Automated Technology for Verification and Analysis (ATVA'12)*, volume 7561 of *LNCS*, pages 317–332. Springer, 2012.

[KNP11] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[KVF+08] S. Khanna, S.S. Venkatesh, O. Fatemieh, F. Khan, and C.A. Gunter. Adaptive Selective Verification. In *IEEE INFOCOM*, pages 529–537, 2008.

[Mas] MasterCard. MasterCard Statement. `http://www.businesswire.com/news/home/20101208005866/en/MasterCard-Statement`.

[MES] MESSI: `http://sysma.lab.imtlucca.it/tools/ensembles/`.

[MMPT13a] Andrea Margheri, Massimiliano Masi, Rosario Pugliese, and Francesco Tiezzi. A Formal Software Engineering Approach to Policy-based Access Control. Technical report, DiSIA, Univ. Firenze, 2013. Available at `http://rap.dsi.unifi.it/facpl/research/Facpl-TR.pdf`.

[MMPT13b] Andrea Margheri, Massimiliano Masi, Rosario Pugliese, and Francesco Tiezzi. Developing and Enforcing Policies for Access Control, Resource Usage, and Adaptation: A Practical Approach. In *WS-FM*, LNCS. Springer, 2013. To appear.

[MS12] Ugo Montanari and Matteo Sammartino. Network conscious -calculus: A concurrent semantics. *Electr. Notes Theor. Comput. Sci.*, 286:291–306, 2012.

[MS13] Ugo Montanari and Matteo Sammartino. A Network-Conscious $\pi$-calculus and Its Coalgebraic Semantics. *To appear in Theor. Comput. Sci.*, 2013.

[NCC] Intelligent robots for improving the quality of life. `http://www.nccr-robotics.ch`.

[OAS12] OASIS XACML TC. eXtensible Access Control Markup Language (XACML) version 3.0 - Candidate OASIS Standard, September 2012.

[OGCD10] Rehan O'Grady, Roderich Groß, Anders Lyhne Christensen, and Marco Dorigo. Self-assembly strategies in a group of autonomous mobile robots. *Autonomous Robots*, 28(4):439–455, 2010.

[PTO+12] Carlo Pinciroli, Vito Trianni, Rehan O'Grady, Giovanni Pini, Arne Brutschy, Manuele Brambilla, Nithin Mathews, Eliseo Ferrante, Gianni Di Caro, Frederick Ducatelle, Mauro Birattari, Luca Maria Gambardella, and Marco Dorigo. Argos: a modular, parallel, multi-engine simulator for multi-robot systems. *Swarm Intelligence*, 6(4):271–295, 2012.

[RD01] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350, 2001.

[Sam] Matteo Sammartino. *A Network-Aware Process Calculus for Global Computing and its Categorical Framework*. PhD thesis, University of Pisa. Submitted for review. Draft available at `http://www.di.unipi.it/~sammarti/PhDThesis_Sammartino.pdf`.

[SV] Stefano Sebastio and Andea Vandin. MultiVeStA: Statistical Model Checking for Discrete Event Simulators. Submitted `eprints.imtlucca.it/1798`.

[SVA05] Koushik Sen, Mahesh Viswanathan, and Gul A. Agha. Vesta: A statistical model-checker and analyzer for probabilistic systems. In Christel Baier, Giovanni Chiola, and Evgenia Smirni, editors, *QEST 2005*, pages 251–252. IEEE Computer Society, 2005.

[Ver04] D. C. Verma. Service level agreements on IP networks. *Proceedings of the IEEE*, 92(9):1382–1388, 2004.

[WEMM12] Martin Wirsing, Jonas Eckhardt, Tobias Mhlbauer, and Jos Meseguer. Design and analysis of cloud-based architectures with klaim and maude. In *9th International Workshop on Rewriting Logic and its Applications (WRLA 2012)*, LNCS. Springer, 2012.