

# ENTRUST: Engineering Trustworthy Self-Managing Systems – Technical Report

## 1 INTRODUCTION

The increasing complexity of software systems combined with the need to develop more versatile, resilient, and reliable systems instigated the vision of self-managing systems, also known as autonomous computing [1, 2]. In this regard, self-managing systems are expected not only to operate with minimal administrator intervention throughout their lifetime, but also to adapt while providing service in response to environment changes, evolving requirements and unexpected failures. As a result, the software engineering research community devoted much effort in developing frameworks, methodologies and architectures that support the engineering of systems enhanced with self-\* capabilities( e.g., self-adaptive, self-managing, self-healing, self-optimising); see [3, 4] for more information.

Despite the research progress achieved since the advent of autonomous computing, the engineering of trustworthy self-managing systems remains a major research challenge. This barrier constrains the applicability of self-managing systems in safety-critical and business-critical applications, as for example, in healthcare, e-commerce, defence and finance. Systems deployed in these application areas should work dependably and must be characterised by high-integrity runtime operation, as defined by NIST [5]: “High integrity software is software that must be trusted to work dependably in some critical function, and whose failure to do so may have catastrophic results, such as serious injury, loss of life or property, business failure or breach of security.”

The software engineering community classified the provision of evidence, i.e., assurances, that a system operates dependably throughout its lifetime among the most important research objectives for self-managing systems [6]. In fact, assurances was amid the research threads highlighted in the most recent roadmaps for self-adaptive systems [7, 8]. We adopt the following definition for assurances: “*Assurances is the provision of evidence that the system satisfies its stated functional and non-functional requirements during its operation in the presence of self- adaptation*” [8].

In this work, we extend the current practices on the provision of assurances for self-managing systems. To this end, we introduce the first tool-supported methodology for assuring the safety of the closed control loops of self-managing systems using established industry practices. Our framework for the ENgineering of TRUstworthy Self-managing sysTems (ENTRUST) integrates:

- an extended version of our approach to developing formally verified monitor-analysis-planning-execution (MAPE) autonomic control loops [9];
- our runtime quantitative verification technique for the analysis stage of these loops [10];

Simos References  
needed

- an industry-adopted approach to arguing security [11] based on the widely used Goal Structuring Notation (GSN) for safety arguments [12].

## 2 Running Example

We will demonstrate the application of ENTRUST using a self-adaptive UUV embedded system, adapted from [13]. UUVs are increasingly used in a wide range of oceanographic and military tasks, including oceanic surveillance (e.g., to monitor pollution levels and ecosystems), undersea mapping, and mine detection. Limitations intrinsic to the environment in which these vehicles operate (e.g., impossibility to maintain UUV-operator communication during missions and high frequency of unexpected changes) require that UUV systems are self-adaptive. These systems are also safety critical (e.g., when used for mine detection and surveillance of ecosystems that should not be impacted) and/or business critical, since UUVs are often expensive equipment that should not be lost during missions.

The self-adaptive system in our study consists of a UUV used to carry out a surveillance and data gathering mission. The UUV is equipped with  $n \geq 1$  on-board sensors that can measure the same attribute of the ocean environment (e.g., water current, salinity or thermocline). When used, the sensors take measurements with different, variable rates  $r_1, r_2, \dots, r_n$ . The probability that each sensor produces measurements that are sufficiently accurate for the purpose of the mission depends on the UUV speed  $sp$ , given by  $p_1, p_2, \dots, p_n$ <sup>1</sup>. For each measurement taken, different amount of energy is consumed, given by  $e_1, e_2, \dots, e_n$ . Finally, the  $n$  sensors can be switched on and off individually (e.g., to save battery power when not required), but these operations consume an amount of energy given by  $e_1^{\text{on}}, e_2^{\text{on}}, \dots, e_n^{\text{on}}$  and  $e_1^{\text{off}}, e_2^{\text{off}}, \dots, e_n^{\text{off}}$ , respectively.

The UUV is required to self-adapt to changes in the observed sensor measurement rates  $r_i$ ,  $1 \leq i \leq n$ , and to sensor failures by dynamically adjusting:

- (a) the UUV speed  $sp$
- (b) the sensor configuration  $x_1, x_2, \dots, x_n$  (where  $x_i = 1$  if the  $i$ -th sensor is on and  $x_i = 0$  otherwise)

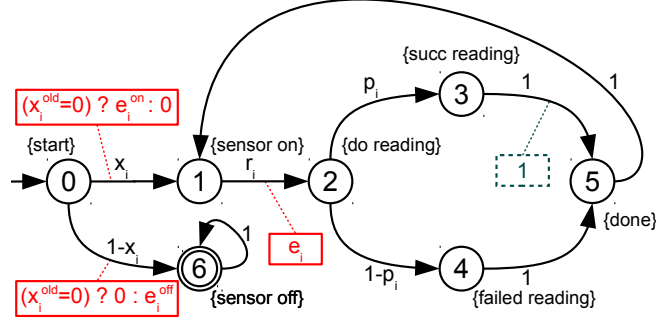
so that the UUV complies with the following requirements at all times:

- R1:** The UUV should take at least 20 measurements of sufficient accuracy for every 10 metres of mission distance.
- R2:** The energy consumption of the sensors should not exceed 120 Joules per 10 surveyed metres.
- R3:** If requirements R1 and R2 are satisfied by multiple configurations, the UUV should use one of these configurations that minimises the cost function

$$cost = w_1 E + w_2 s^{-1}, \quad (1)$$

---

<sup>1</sup> This information can be extracted from the technical specification of sensors; for example, see <http://www.ashtead-technology.com/rental-equipment/teledyne-rdi-600khz-navigator/>



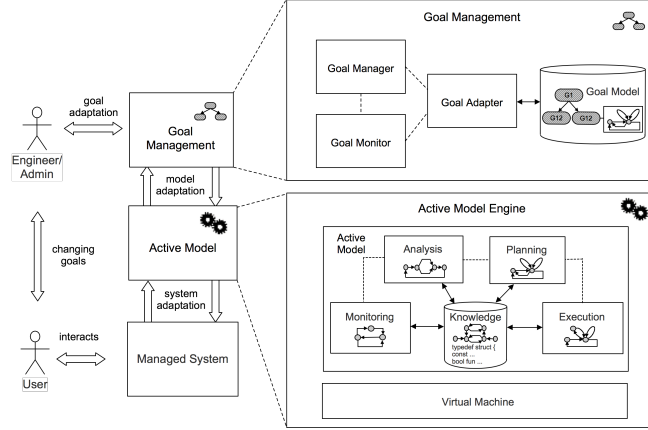
**Fig. 1.** CTMC model  $M_i$  of the  $i$ -th UUV sensor

where  $E$  represents the energy consumed by the sensors to survey a 10m mission distance, and  $w_1, w_2 > 0$  represent weights that reflect the relative importance of carrying out the mission with reduced battery usage and completing the mission faster.

*Example 1.* Figure 1 depicts the CTMC model  $M_i$  of the  $i$ -th sensor of the UUV from our running example. The CTMC starts in state 0 and transitions in state 1 if the sensor is switched on ( $x_i = 1$ ) or in state 6 otherwise ( $x_i = 0$ ). With rate  $r_i$ , a measurement is performed, as indicated by the transition between states 1 and 2. The measurement is “sufficiently accurate” with probability  $p_i$  and the transition between states 2 and 3 is taken, whereas the transition between states 2 and 4 is enabled in case of an inaccurate measurement. An active sensor continues taking measurements while it is active, as modelled by the transition between states 5 and 1. The CTMC model is augmented with two cost/reward structures, whose non-zero elements are shown in Fig. 1 in rectangular boxes, and dashed rectangular boxes, respectively. The former, “energy” structure associates the energy used to switch the sensor on (i.e.,  $e_i^{\text{on}}$ ) and off (i.e.,  $e_i^{\text{off}}$ ) and to perform a measurement (i.e.,  $e_i$ ) with the CTMC transitions that model these events. Note that the ternary expression “*condition*?*a*:*b*” was used to indicate that the energy  $e_i^{\text{on}}$  is consumed only if the previous state of the sensor was “off”, i.e., if  $x_i^{\text{old}} = 0$ , whereas the energy  $e_i^{\text{off}}$  is used only if the opposite is true ( $x_i^{\text{old}} = 1$ ). As concerns the latter, “measurement” cost/reward structure, it associates a reward of 1 with the transition that corresponds to an accurate measurement.

### 3 ActivFORMS

ActivFORMS is a formal approach for self-adaptation that uses an integrated formal model of the adaptive components and knowledge models. The formal model is executed directly by a virtual machine to realize adaptation, hence called active model. ActivFORMS approach distinguishes itself for existing approaches in two ways. First, the formally verified model of the complete feedback



**Fig. 2.** ActivFORMS approach

loop is directly executed by the virtual machine, which guarantees the verified adaptation goals at runtime. As the active model is directly executed by ActivFORMS, the approach does not require coding. Second, ActivFORMS supports dynamic change of the active model. A new feedback loop model can be deployed at runtime to meet new or changing goals.

Fig. 2 shows the primary modules of the ActivFORMS. The approach is in line with the three layered reference model for self-adaptive systems proposed by Kramer and Magee [14]. The managed system realizes the domain functionality for users. The active model consists of two parts: an integrated formal model that realize a feedback loop, i.e., the active model, and a virtual machine that can execute the active model. The active model monitors the managed system through probes and adapt the managed system through effectors.

ActivFORMS supports feedback loops modeled using networks of timed automata [15]. A timed automaton is a finite-state machine that models a behavior, extended with clock variables. Automata can synchronize through channels. There are two types of channels, binary channels and broadcast channels. For a binary channel, a sender  $x!$  can synchronize with a receiver  $x?$  through a signal. If there are multiple receivers  $x?$  then a single receiver will be chosen non-deterministic. The sender  $x!$  will be blocked if there is no receiver. A broadcast channel sends a signal to all the receivers, and if there is no receiver, the sender will not be blocked. Behavior specifications can be complemented with expressions specified in a C-like language to define data structures (struct concept) and functions. Goals can be expressed in timed computation tree logic expressions (TCTL). TCTL expressions describe the state and path formulae that can be verified, such as reachability (a system should/can/cannot/... reach a particular state or set of states), liveness (something eventually will hold), etc. We use Uppaal [16], a model checking tool that supports modeling of behaviors and verification of properties.

The goal management layer handles adaptation issues that cannot be performed by the current active model. The goal management consists of four parts, i.e., goal model, goal monitor, goal adapter and goal manager. The goal model represents the adaptation goals. ActivFORMS uses AND-OR tree-based goal model to specify goals. The goals can be expressed as boolean expressions, and the goals at the bottom level of each subtree have associated models to realize adaptations. The goal monitor monitors the status of the goals and if any goal is violated it notifies the goal adapter. The goal adapter is the heart of goal management. When the goal adapter is signaled by the goal monitor about a change of goals, it consults the goal model and search for a matching model that satisfies the changing situation. If the model associated with the changing goal differs from the currently deployed model, the goal adapter starts updating the current model with the new model at the virtual machine. If the model does not differ no further action is required. The goal manager offers support for three primary functions: inspecting the active model and its ongoing execution, monitoring and updating goals, and updating the goal model. In our current implementation, the ActivFORMS User Interface connects with the goal managers of the different nodes of the system. The user interface enables the system admin (or engineer) to perform the functions of the goal managers remotely.

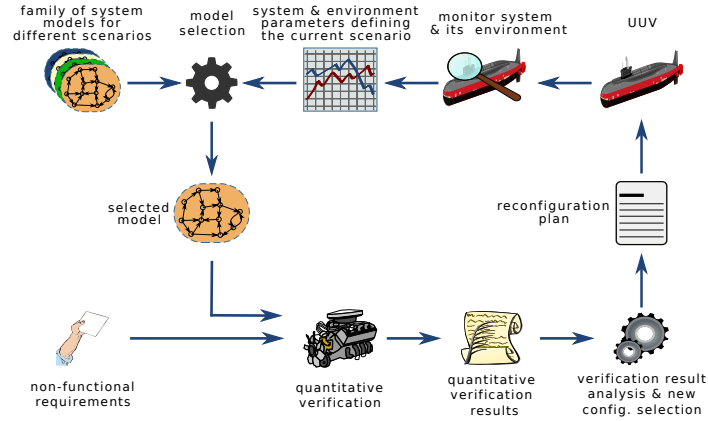
## 4 Runtime Quantitative Verification

*Runtime quantitative verification* (RQV) [10] is an approach to implementing the closed control loop of self-adaptive systems in which the adaptation decisions are driven by the continual verification of stochastic models. RQV was introduced in [17, 18] and further refined in [19–21].

Figure 3 depicts the adaptation workflow of an RQV-based self-adaptive system. The approach involves monitoring the system (e.g., the UUV) and its environment continuously, to identify relevant changes and quantify them using fast on-line learning techniques. When these changes are significant and/or periodically, an RQV step is triggered, to identify which scenario the system operates in, and to select the associated model from a family of system models (i.e., a *parametric model*) that correspond to different such scenarios. The selected model is then analysed using quantitative verification, to identify and/or predict violations of QoS requirements such as response time, availability and cost. When QoS violations are identified or predicted, the verification results support the selection of a new set of values for the configurable system parameters, so that enforcing this new configuration is *guaranteed* to restore or maintain compliance with QoS requirements, respectively.

## 5 Implementation

This section will explain how we achieved our goal.



**Fig. 3.** RQV-based self-adaptive system

## 5.1 Managed System

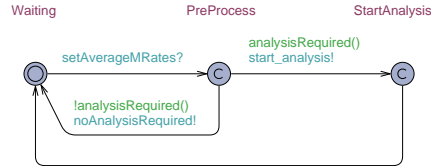
Details about managed system

## 5.2 Managing System

In this section, we briefly describe the managing system automata.

**Monitor** Monitor automaton receives the sensors rates from probes. After that, the monitor automaton checks that whether received sensor rates are equal to previously measured sensor rates. If both rates are equal, then there is no need to analyze because an analysis already have been applied to these sensor rates. If sensor rates are different then analysis is required and the monitor automaton will invoke the analyze automaton by sending an *start\_analysis!* signal as shown in Fig. 4.

**Analyze** Analyze automaton takes the current measured rates of sensors and try to find the best configuration which fulfills the requirement R1, R2 and R3. Fig. 5 shows the analyze automaton of feedback loop. The analyze automaton starts the analysis when received the signal *start\_analysis?* from monitor automaton. After receiving the signal from monitor automaton, the analyze automaton sends the sensor rates to RQV by sending the signal *calcProbability*. When the RQV receive the signal it creates a set of possible configurations and send back to the analyze automaton through *recvProbability?* channel. After that the analyze automaton analyzes the set of configurations and try to find a best configuration which holds the R1, R2 and R3. Once the best configuration is found, the analyze automaton compares the current configuration of UUV to the best configuration. If both configurations are equal, then there is no need to plan anything. If the best configuration is not equal to the current configuration of UUV, the analyze

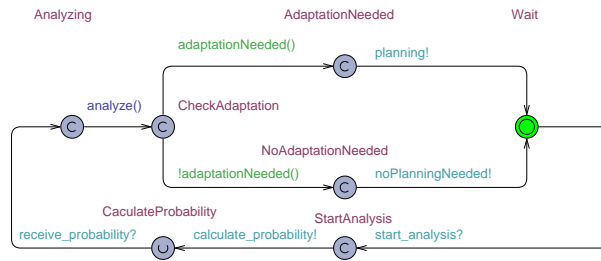


**Fig. 4.** Monitor automaton of UUV

automaton recognize that this is the time to do an adaptation, and send the signal *planning!* to invoke the planner. Fig. 5 shows the analyze automaton of the UUV.

**Plan** Plan automaton creates the plans for adaptation. The plan automaton starts the planning when the signal *planning?* is received. The plan automaton starts first with creating the plan steps for sensor configuration, i.e., whether a sensor should be turned on or off. To do that, the plan automaton checks the current sensor configuration with the best configuration for each sensor to determine whether there is a need to turn on the sensor or off. If the sensor configuration is same as in the best configuration, then there is no need to create a plan step. Otherwise, if sensor configurations are not matched then the plan automaton creates the plan step, according to the best configuration. When all sensors are checked, then the plan automaton checks the current speed of the UUV with the UUV speed in best configuration. Similarly, if the current UUV speed doesn't match with the speed in best configuration, the plan automaton creates a new plan step to change the speed of UUV, otherwise the plan automaton continues. After checking the UUV speed, the plan automaton invokes the execute automaton by sending the *execute!* signal. Fig. 6 shows the plan automaton of the UUV.

**Execute** The execute automaton starts by the signal *execute?* sent from the plan automaton. After receiving the signal the execute automaton goes through



**Fig. 5.** Analyze automaton of UAV

all the plan steps one by one and execute them. The plan automaton is presented in the Fig. 7.

### 5.3 Probes & Effectors

If necessary code of probes & effectors

### 5.4 Offline Assurances

The provision of partial offline assurances

### 5.5 Online Assurances

The perpetual provision of assurances online. How we are using both technologies together.

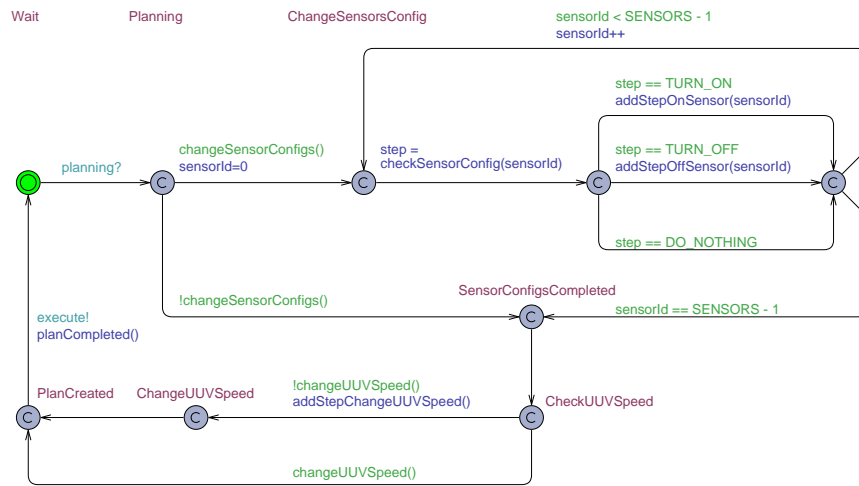


## 6 Methodology

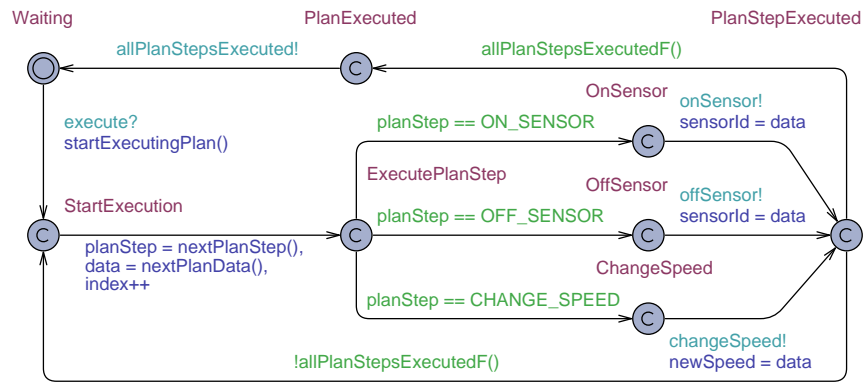
### References

1. J. Kephart and D. Chess, “The vision of autonomic computing,” *Computer*, vol. 36, no. 1, pp. 41–50, 2003.
2. A. Ganek and T. A. Corbi, “The dawning of the autonomic computing era,” *IBM Systems Journal*, vol. 42, no. 1, pp. 5–18, 2003.
3. M. Salehie and L. Tahvildari, “Self-adaptive software: Landscape and research challenges,” *ACM Trans. Auton. Adapt. Syst.*, vol. 4, pp. 14:1–14:42, May 2009.
4. M. C. Huebscher and J. A. McCann, “A survey of autonomic computing-degrees, models, and applications,” *ACM Comput. Surv.*, vol. 40, pp. 7:1–7:28, Aug. 2008.
5. L. I. D. R. Wallace and D. R. Kuhn, “High integrity software standards and guidelines,” *National Institute of Standards and Technology*, vol. NIST SP 500-204, September 1992.
6. B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle, “Software engineering for self-adaptive systems: A research roadmap,” ch. Software Engineering for Self-Adaptive Systems: A Research Roadmap, pp. 1–26, Berlin, Heidelberg: Springer-Verlag, 2009.
7. R. de Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. M. Goeschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, M. Litoiu, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezzè, C. Prehofer, W. Schäfer, R. Schlichting, B. Schmerl, D. B. Smith, J. P. Sousa, G. Tamura, L. Tahvildari, N. M. Villegas, T. Vogel, D. Weyns, K. Wong, and J. Wuttke, “Software engineering for self-adaptive systems: A second research roadmap,” in *Software Engineering for Self-Adaptive Systems II* (R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, eds.), vol. 7475 of *Lecture Notes in Computer Science*, pp. 1–32, Springer Berlin Heidelberg, 2013.
8. R. de Lemos, D. Garlan, C. Ghezzi, and H. Giese, “Software Engineering for Self-Adaptive Systems: Assurances (Dagstuhl Seminar 13511),” *Dagstuhl Reports*, vol. 3, no. 12, pp. 67–96, 2014.
9. M. U. Iftikhar and D. Weyns, “Activforms: Active formal models for self-adaptation,” in *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014, (New York, NY, USA), pp. 125–134, ACM, 2014.
10. R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola, “Self-adaptive software needs quantitative verification at runtime,” *Commun. ACM*, vol. 55, pp. 69–77, Sept. 2012.
11. C. B. Weinstock, H. F. Lipson, and J. Goodenough, “Arguing security - creating security assurance cases, us department of homeland security,” 2007. [Online] <https://buildsecurityin.us-cert.gov/articles/knowledge/assurance-cases/arguing-security—creating-security-assurance-cases>.
12. T. Kelly and R. Weaver, “The goal structuring notation – a safety argument notation,” in *Proc. of Dependable Systems and Networks 2004 Workshop on Assurance Cases*, 2004.

13. S. Gerasimou, R. Calinescu, and A. Banks, "Efficient runtime quantitative verification using caching, lookahead, and nearly-optimal reconfiguration," in *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014, (New York, NY, USA), pp. 115–124, ACM, 2014.
14. J. Kramer and J. Magee, "Self-Managed Systems: an Architectural Challenge," *FOSE*, 2007.
15. R. Alur and D. Dill, "Automata for modeling real-time systems," in *Automata, Languages and Programming* (M. Paterson, ed.), vol. 443 of *Lecture Notes in Computer Science*, pp. 322–335, Springer Berlin Heidelberg, 1990.
16. G. Behrmann, R. David, and K. G. Larsen, "A tutorial on uppaal," pp. 200–236, Springer, 2004.
17. R. Calinescu and M. Z. Kwiatkowska, "Using quantitative analysis to implement autonomic IT systems," in *Proceedings of the 31st International Conference on Software Engineering, ICSE 2009*, pp. 100–110, IEEE Computer Society, 2009.
18. I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli, "Model evolution by run-time parameter adaptation," in *Proc. 31st International Conference on Software Engineering (ICSE09)*, (Los Alamitos, CA, USA), pp. 111–121, IEEE Computer Society, 2009.
19. R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli, "Dynamic QoS management and optimization in service-based systems," *IEEE Trans. Softw. Eng.*, vol. 37, pp. 387–409, 2011.
20. A. Filieri, C. Ghezzi, and G. Tamburrelli, "Run-time efficient probabilistic model checking," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*, (New York, NY, USA), pp. 341–350, ACM, 2011.
21. K. Johnson, R. Calinescu, and S. Kikuchi, "An incremental verification framework for component-based software systems," in *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '13, (New York, NY, USA), pp. 33–42, ACM, 2013.



**Fig. 6.** Plan automaton



**Fig. 7.** Execute automaton