ascens

www.ascens-ist.eu

ASCENS

Autonomic Service-Component Ensembles

D5.2: Second Report on WP5

Verification Techniques for SCs and SCEs (first version)

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), Saddek Bensalem (UJF-Verimag), Christian von Essen (UJF-Verimag), Nora Koch (LMU), Jan Kofron (CUNI)**

Reporting Period: **2**
Period covered: **October 1, 2011 to September 30, 2012**
Submission date: **November 12, 2012**
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

This document summarizes the work performed in Year 2 targeted to develop new techniques and theories to support the design and the implementation of correct and reliable service components (SC) and service component ensembles (SCE). We have worked in several different directions in order to get closer to our goal. Our contributions can be grouped into the following main directions: (1) We have presented a new and flexible approach to repair reactive programs with respect to a specification. (2) We have developed a semi-symbolic algorithm for synthesizing controllers in a stochastic environment, implemented as an add-on to the probabilistic model checker PRISM. (3) We have presented a new method for generating linear invariants for SC. In particular we developed an incremental approach that allows discovering and reusing invariants that have been already been computed on subparts of the model. These new techniques have been implemented in D-Finder, a tool for checking deadlock freedom. (4) We propose an approach to make the specification of access control policies accessible to people not necessarily familiar with formal languages. (5) We propose a formal pattern-based approach to study defense mechanisms against DoS attacks. We enhance pattern descriptions with for- mal models that allow the designer to give guarantees on the behavior of the proposed solution. (6) Finally, we developed and extension of GMC with support of C++. Also, we have created a local version of the jDEECo framework.

Contents

1 Introduction **5**

2 Advancements on Verification and Design of Service Components **6**
2.1 Program Repair Revisited 7
2.2 Semi-Symbolic Computation of Efficient Controllers in Probabilistic Environments . 9

3 Advancements on Verification of Service Component Ensembles **11**

4 Advancements on Security Policies and Access Control **13**
4.1 Model-Driven Development of Access Control Policies 15
4.2 Stable Availability under Denial of Service Attacks through Formal Patterns 18

5 Advancements on Verification of SC Implementation Compliance with High-level Specification **19**
5.1 GMC 19
5.2 JPF 19
5.3 Development of methods 20

6 Conclusion and Summary of the Main Achievements **20**

7 Next Step and Long-Term Technical goals **21**

1 Introduction

Our goal in Workpackage 5 is to develop new techniques and the underlying theories to support the design and the implementation of correct and reliable service components (SCs) and service component ensembles (SCEs). We are working in several directions. The first one deals with correctness on the level of a service component, considering in particular non-functional properties like resource-awareness. The second deals with correctness of ensembles of service components mainly focusing on constructive techniques. Given the particular importance of security in ensembles, the third will address the problem of building secure ensembles. Finally, we work on techniques to check if a SCs implementation complies with a high-level specification. In year 2, we have worked on the four tasks:

1. In Task T5.1 *Verification and Design of Service Component*, we worked on:
   - Program Repair Revisited. In this work we developed a new and flexible approach to repair reactive programs with respect to a specification. The specification is given in linear-temporal logic. Like in previous approaches, we require that a repaired program satisfies the specification and is syntactically close to the faulty program. In addition our approach also allows the user to ask for a program that is semantically close by enforcing that a specific subset of the correct traces is preserved. Our approach is based on synthesizing a program producing a set of traces that stays within a lower and an upper bound. We provide an algorithm to decide if a program is repairable with respect to our new notion and synthesize a repair if one exists. We analyze several ways to choose the set of traces to leave intact and show the boundaries they impose on repairability.
   - Semi-Symbolic Computation of Efficient Controllers in Probabilistic Environments. In this work we obtained a semi-symbolic algorithm for synthesizing controllers in a stochastic environment, implemented as an add-on to the probabilistic model checker PRISM. The user specifies the environment and the controllable actions using a Markov Decision Process (MDP), modeled in the PRISM language. Controller efficiency is defined with respect to a user- specified assignment of costs and rewards to the controllable actions. An optimally efficient strategy minimizes the ratio between the encountered costs and rewards. At the core of the implementation is the first semi-symbolic algorithm based on a recently developed strategy improvement algorithm for MDPs with ratio objectives. We show the effectiveness of our implementation using a set of benchmarks.
2. In Task T5.2 *Verification of Service Component Ensembles*, we worked on incremental generation of linear invariants for Component Ensemble. In this work we developed a new method for generating invariants. Linear invariant generation has been extensively considered as an effective verification method for concurrent systems. However, none of the existing work on the topic strongly exploits the structure of the system and the algebra that defines the interactions between its components. The objective of this work is to extend and propose new techniques dedicated to the computation of linear interactions invariants, i.e., invariants that are described by linear constraints and that relate states of several components in the system. In particular, we propose an incremental approach that allows discovering and reusing invariants that have already been computed on subparts of the model. Those new techniques have been implemented in D-finder and evaluated on several case studies. The experiments show that our approach outperforms classical techniques on a wide range of models.
3. In Task T5.3 *Security Policies and Access Control*, we worked on:
   - *Model-Driven Development of Access Control Policies*. In this work, we developed a model-driven approach that supports the generation of access control policies written in the

OASIS standard language XACML and in a formally founded language (called FACPL). Access control policies are modeled using an extension of the UWE modeling language for web applications. XACML is used for expressing and enforcing policy-based authorizations and FACPL is appropriate for applying diverse reasoning techniques. The model-driven approach is implemented by a tool chain that mainly comprises the UML CASE tool MagicUWE and the model transformations UWE2XACML and XACML2FACPL.

- *Stable Availability under Denial of Service Attacks through Formal Patterns*. In this work, 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. Then 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. As a main result we show, by statistical model checking with the PVeStA tool, that the composition of both patterns yields a new improved pattern which guarantees stable availability at a reasonable cost.

4. For Task T5.4 *Verification of SCs implementation compliance with high-level specification*, we present an extension of GMC with support of C++, which is to be completed in the near future. We have created a local version of the jDEECo framework and adapted it to be compliant to the input language of JPF and discovered the potential practical issues with model checking it. As to the methods, we have worked on techniques for deducing ownership of objects; this information can be used for making partial order reduction more powerful, i.e., to handle larger state spaces.

This deliverable summarizes the work performed in Year 2. In the next sections, we give a detailed description of each contribution separately, and a conclusion and long-term technical goals.

2 Advancements on Verification and Design of Service Components

As explained in Deliverable JD2.2, the design flow considered in ASCENS is an iterative process including design steps and validation steps. Validation corresponds to a phase during which the behavior of the designed system is checked against its expected one. It can be achieved by testing, that is, considering specific input or scenarios for which the system is simulated or executed, or by verification, that is, using automatic or semi-automatic tools for (mathematically) proving the correctness of the system. When misbehaviors are discovered, the system must be fixed and validated again. Hopefully, this iterative process converges and produces a correct system.

Another approach is to automatically modify the misbehaving sytem in order to enforce correct behavior. The two methods presented in Sections 2.1 and 2.2 fall in this category. Using these methods, we expect to automate and accelerate some steps of the ASCENS design flow. The first method considers a program and an LTL formula φ from which it tries to compute a fix for the program, that is, enforcing φ. The second one proposes to automatically generate the controller for a system in order to optimize its average performance based on a cost/reward model, in a probabilistic environment.

2.1 Program Repair Revisited

Debugging a program is often a difficult and tedious task: a programmer has to find the bug, localize its cause, and repair it. Model checking [CE81, QS82] has been successfully used to expose bugs in a program. There are several interesting approaches [CGMZ95, ELLL01, RS04, ZH02, JRS02, GV03, BNR03, RR03] to explain the possible cause of an error. We are interested in addressing the last debugging step by automatically providing a bug-fix. Automatic program repair takes a program and a specification and searches for a correct program that satisfies the specification and is syntactically close to the original program [BEGL99, JGB05, EKB05, GBC06, JM06, CMB08, SDE08, VYY09, CTBB11]. We are inspired by the work of Jobstmann et al. [JGB05, JSGB12] and Chandra et al. [CTBB11]. Jobstmann et al. suggest to repair a program based on a game derived from the program and a formal specification given in Linear-Temporal Logic (LTL) [Pnu77]. The authors allow the repair procedure to replace an arbitrary component (e.g., an assignment or an if-condition) and require that the repaired program satisfies the formal specification. This approach has the benefit that no specific fault model needs to be defined and that expressive repair statements can be found. However, the automatic repair tool has the freedom to change the original program substantially, and there is no way to restrict this freedom.

Chandra et al. follow a different approach, they search in the space of all edits of a program for one edit that repairs failing tests without breaking any passing tests. We believe that the idea of not breaking correct behaviors is essential for automatically repairing programs. We show how to generalize this idea to reactive programs with specifications and present the first repair approach that constructs repairs that are also semantically close to the original program. The key benefits of our approach are: (i) One can adjust how close the original program and the repair should be. (ii) The approach is less sensitive than previous approaches to the set of allowed program modifications. E.g., the approach in [JGB05] constructs degenerated programs if given too much freedom in modifying the program. Our approach is less likely to construct degenerated programs because it preserves correct behaviors and their properties. (iii) Finally, since it preserves the core behavior of the program, the need for a complete specification is reduced.

Let us motivate the problem with an example. Assume that we want to develop a sensor-driven traffic light system for a crossing of two streets. The system has two sets of lights (called light1 and light2) and two sensors (called sensor1 and sensor2), one set of lights and one sensor for each street entering the crossing. By default both lights are red. If a sensor detects a car, then the corresponding lights should change from red to yellow to green and back to red. As starting point we are given the implementation shown in Figure 1. It behaves as follows: for each red light, the system checks if the sensor is activated (Lines 12 and 18). If so, this light becomes yellow in the next step, followed by a green phase and a subsequent red phase. Assume we require that our implementation is safe, i.e., the two lights are never green at the same time. In LTL, this specification is written as $\varphi = \mathbf{G}(\mathsf{light1} \neq \mathsf{GREEN} \vee \mathsf{light2} \neq \mathsf{GREEN})$, where **G** denotes the "always" operator. The current implementation clearly does not satisfy this requirement: if both sensors signal a car initially, then the lights will simultaneously move from red to yellow and to then to green, thus violating the specification.

Let us try to repair the given implementation. Following the approach in [JGB05] we introduce a non-deterministic choice into the program and then use a synthesis procedure to select among these options in order to satisfy the specification. For instance, we replace Line 12 (in Figure 1) by **if**(?) and ask the synthesizer to construct a new expression for ? using the input and state variables. The synthesizer aims to find a simple expression s.t. φ is satisfied. In this case one simple admissible expression is **false** because replacing Line 12 with **if** (**false**) ensures that the modified program satisfies specification φ. While this suggested repair is correct, it is very unlikely to please the programmer because it repairs "too much": it modifies the behavior of the system also on input traces on which

```
1  typedef enum {RED, YELLOW, GREEN} traffic_light;
2  module Traffic (clock, sensor1, sensor2, light1, light2);
3    input clock, sensor1, sensor2;
4    output light1, light2;
5    traffic_light reg light1, light2;
6    initial begin
7      light1 = RED;
8      light2 = RED;
9    end
10   always @(posedge clock) begin
11     case (light1)
12       RED: if (sensor1) // Repair : if(sensor1 & !(light2 == RED & sensor2))
13               light1 = YELLOW;
14       YELLOW: light1 = GREEN;
15       GREEN: light1 = RED;
16     endcase // case (light1)
17     case (light2)
18       RED: if (sensor2)
19                light2 = YELLOW;
20       YELLOW: light2 = GREEN;
21       GREEN: light2 = RED;
22     endcase // case (light1)
23   end // always (@posedge clock)
24 endmodule // traffic
```

Figure 1: Implementation of a traffic light system and a repair.

the initial implementation was correct. We believe it is more desirable to follow the idea of Chandra et al. [CTBB11] saying that a repair is only allowed to change the behavior on incorrect execution. In this case the repair suggested above would not be allowed because it changes the behavior on correct traces.

Exact repair. In the proposed method, programs are modeled as finite input/output state machines defined in terms of labeled transition systems. Given an LTL formula φ representing a specification and a program (i.e. a machine) M, we say that the machine M' is an *exact repair* for M if (i) M' behaves like M for all traces satisfying φ and (ii) if M' fulfills φ, which is formally expresses by:

$$\mathrm{L}(M) \cap \mathrm{L}(\varphi) \subseteq \mathrm{L}(M') \subseteq \mathrm{L}(\varphi),$$

where L is an operation associating to a machine or an LTL formula its corresponding accepted language. Note that the first inclusion defines the behavior of M' on all input words to which M responds correctly according to φ. Figure 2 illustrates this definition. The two circles depict $\mathrm{L}(M)$ and $\mathrm{L}(\varphi)$. A repair has to (i) cover their intersection, which we depict with the striped area in the picture, and (ii) lie within $\mathrm{L}(\varphi)$. One such repair is depicted by the dotted area on the right.

Following our definition the repair suggested for example of Figure 1 is not a valid repair anymore. The reason is that the original implementation responds correctly, e.g., to the input trace in which sensor1 is always high and sensor2 is always low, but the repair produces different outputs. In fact the initial implementation behaves correctly on any input trace on which sensor1 and sensor2 are never high simultaneously. So, any exact repair should include this input/output traces. An exact

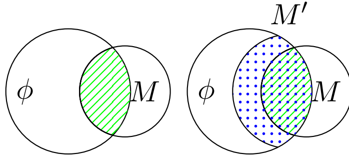

Figure 2: Representation of *exact* repair.

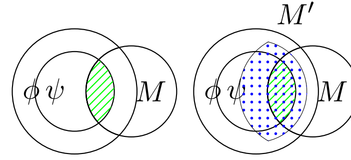

Figure 3: Representation of (relaxed) repair.

repair replaces **if** (sensor2) by **if** (sensor1 & !(light2 == RED & sensor2)). This repair retains all correct traces while avoiding the mutual exclusion problem.

Relaxed repair. An exact repair does not always exist [vEJ12b]. The reason is that since the behavior of the repaired machine M' has to mach the one of M for correct traces, M' has to mimic M as long as M has a chance to satisfy φ. For some machine M and formula φ, it may happen that M has always a chance to satisfy φ. In this case $M' = M$ incorrectly repairs M with respect to φ. In order to find a repair, we have to relax the requirement that M' has to behave like M on all correct traces, leading to the following definition. We say M' is a repair of M with respect to ψ and φ if M' behaves like M on all traces satisfying ψ, and if M' fulfills φ. That is, M' is a repair constructed for M iff:

$$\mathrm{L}(M) \cap \mathrm{L}(\psi) \subseteq \mathrm{L}(M') \subseteq \mathrm{L}(\varphi).$$

Figure 3 provides a graphical representation of this definition.

We have shown that the problem of finding a repair can be reduced to the classical synthesis problem of [Chu63], which can be solved using standard techniques for building a repair if exists. We also provided a necessary and sufficient condition for existance of such a repair M' for a machine M and formula ψ and φ. We also studied different approaches for choosing the preserving part ψ of the behavior of a program M, and discussed the reason for repair failures. We plan to improve this according to two orthogonal directions. The first one increases the computational power of a repair machine. The second direction studies relaxed notion of set-inclusion in order to express how close two machines are in a quantitative way, e.g. by measuring how often they differ.

2.2 Semi-Symbolic Computation of Efficient Controllers in Probabilistic Environments

Efficiency is a central point in controller design. In many settings, the controller needs to make a trade-off between competing quantities. For instance, consider a production line in which the speed, i.e., the number of produced units, can be adjusted. A controller that produces as many units as possible seems preferable. However, running the line in a faster mode increases the power consumption and the probability to fail, resulting in higher repair costs. Therefore, it is natural to ask for an *efficient* controller, i.e., a controller that minimizes the power and repair costs per produced unit.

We developped semi-symbolic techniques for not only analyzing how efficient a controller is, but also for automatically constructing the *most efficient* controller for a given environment. Efficiency is defined as the ratio between a given cost model and a given reward model [vEJ12a]. This choice is inspired by the idea that an efficient system has to balance between the time or effort it uses versus the intended task. For instance, consider an automatic gear-shifting unit (ACTS) that optimizes its behavior for a given driver profile. The goal of the ACTS is to optimize the fuel consumption per kilometer (l/km), a commonly used unit to quantify efficiency. In order to be most efficient, the system has to maximize the speed (given in km/h) while minimizing the fuel consumption (measured in liters per hour, i.e., l/h) for the given driver profile. If we take the ratio between the fuel consumption (the costs) and the speed (the reward), we obtain l/km, the desired measure. Objectives of this nature have

also been shown to be appropriate in other contexts, such as the ratio between energy consumption and latency, and the ratio between detected collisions and failed transmissions as a measure of efficiency in MAC protocols [YBK10].

We also implemented the method allowing the user to specify a probabilistic model of the system to control, together with an assignment of costs and rewards to each possible action that the controller can choose. The output of the tool is a controller that optimizes the expected ratio between the accumulated costs and rewards. Our implementation is semi-symbolic: it combines both symbolic (Binary Decision Diagram-based) and explicit-state techniques. It takes the form of an add-on to PRISM [KNP11], a probabilistic model checker that supports verification of Markov chains, Markov decision processes (MDPs) and probabilistic timed automata. PRISM provides model checking of a variety of quantitative properties, including some reward-based measures, but has no support for ratio objectives, which require rather different techniques. Ratio-based measures are a useful class of properties that are not expressible with multi-objective MDP model checking [FKN+11a]. As a side-effect, PRISM can now also handle MDPs with the classical average objective.

The main contribution is the semi-symbolic algorithm we proposed for finding optimal strategies for MDPs with ratio objectives (Ratio-MDPs) [vEJ12a]. To encode MDPs symbolically, we use Multi-Terminal Binary Decision Diagrams (MTBDDs) [FMY97],, which are efficient data structures, generalizing Binary Decision Diagrams (BDDs) [Bry92], to represent functions from finite domains to finite ranges. Given a trace ρ (i.e., an infinite sequence of state-action pairs ρ_i) of an MDP, the ratio objective expresses the ratio between the accumulated costs and the accumulated rewards, defined as:

$$\lim_{l \to \infty} \lim_{n \to \infty} \frac{\sum_{i=l}^{n} costs(\rho_i)}{1 + \sum_{i=l}^{n} rewards(\rho_i)}.$$

We aim for efficient controllers that minimize the costs per obtained reward, i.e., optimal strategies that minimize the expected ratio value.

Last year we improved our technique so that there is no need for unichains MDPs, which was a major bottleneck of algorithms we proposed previously in [Bry92], leading to a drastic increase of performance. The proposed algorithm proceeds as follows: first, it partitions the Ratio-MDP into end-components [dA97], then it computes an optimal strategy for each end-component, and finally merges these strategies. For the first and the last step, we use algorithms developed for MDPs with average objectives (Average-MDP), a well-studied objective that can be seen as a special case of the ratio objective. We have implemented symbolic versions of these algorithms. The computation of the optimal strategy in an end-component is based on a sequence of reductions of the Ratio-MDP together with a strategy to an Average-MDP. To solve the induced Average- MDPs, we leverage the work of Wimmer et al. [WBB+10]. We have reimplemented their semi-symbolic (symblicit) algorithm for Average-MDPs. Once we have computed a strategy for the entire Ratio-MDP, we transform it into a PRISM-like format to make it readable by the user.

Our current implementation is fully integrated into the PRISM model checker, providing easy-to-use tool support for the ratio optimization criterion. We also added human-readable output of strategies to PRISM. We compared our implementation to existing state of the art techniques. Figure 4 shows the results of our implementation on various benchmarks. Examples *pump3-6* model the water-pump system described in [vEJ12a]. In experiments *phil6-10*, we use Lehmanns formulation of the dining philosophers problem [LR81]. We measured the amount of time a philosopher spends. This model is effectively a mean-payoff because we have a cost of one for each step. We use this experiment to compare our implementation to [WBB+10]. We are several orders of magnitude faster. We attribute the increase in speed to good initial strategy.

We also modeled an automatic clutch and transmission system (*acts*). Each state consists of a driver/traffic state (waiting in front of a traffic light, breaking because of a slower car, free lane),

Benchmark name	#States	Time (in sec)	RAM (in MB)
pump3	386	0.9	112
pump4	1560	5.6	150
pump5	5904	20.5	236
pump6	21394	96.8	326
rabin3	27766	5.2	199
rabin4	668836	104.6	537
zeroconf	89586	2948.7	608
acts	1734	1.6	159
phil6	917424	1.2	181
phil7	9043420	1.9	262
phil8	89144512	2.6	295
phil9	878732012	3.3	287
phil10	8662001936	4.3	303
power1	8904	0.415	89.9
power2	8904	n/a	85

Figure 4: Experimental results.

current gear (1-4) and current motor speed (100 - 500 RPM). We modeled the change of driver state probabilistically, and assumed that the driver wants to reach a given speed (50 km/h). Given this driver and traffic profile, the transmission rates and the fuel consumption based on motor speed, we synthesized the best points to shift up or down.

In *power1-2*, we used the examples from [NPK$^+$05, FKN$^+$11b], which the authors use to analyze dynamic power management strategies. Our implementation allows solution of optimization problems that are not possible with either [NPK$^+$05] or the multiobjective techniques in [FKN$^+$11b]. For example, in *power1* we ask the question What is the best average power consumption per served request. In *power2*, we ask for the best-case power consumption per battery lifetime, i.e., we ask for how many hours a battery can last.

3 Advancements on Verification of Service Component Ensembles

We propose a compositinal and incremental verification method that can be applied to ensembles having a static architecture, that is, having a fixed, pre-established, and known set of components and interactions. This method implemented in the BIP framework is integrated in the general ASCENS design flow by the translation of SCEL specifications into BIP models, as explained in Deliverable JD2.2.

Component-based design confers numerous advantages, in particular, increased productivity through reuse of existing components. Nonetheless, establishing the correctness of ensembles of components remains an open issue. In contrast to other engineering disciplines, software and system engineering badly ensures predictability at design time. Consequently, a posteriori verification as well as empirical validation are essential for ensuring correctness. Monolithic verification [JPJ82, CGP99] of component-based systems is a challenging problem. It often requires computing the product of the components by using both interleaving and synchronization. The complexity of the product system is often prohibitive due to state explosion. A solution to this problem is to generate an invariant that is an abstraction of the state-space of the system.

We observed that most of the existing work on generating invariants for component-based systems are too general and do not strongly exploit the structure of the system and the algebra that defines the interactions between its components. In a series of recent works [BBNS08, BBNS09, BBL$^+$10],

we proposed novel approaches and the DFINDER tool [BGL+11] for generating invariants for systems described in the BIP framework [BIP]. Our techniques start by building invariants for individual components, which can be done with any existing approach for invariant generation on sequential programs. The novel concept in DFINDER is that the invariant for the overall system is then obtained by glueing this set of individual invariants with another one that is an abstraction of the algebra used to define the interactions between the components. By doing so, one avoids building huge parts of the state-space before generating the invariant. One of the major advantages of our approach is that it allows for the development of incremental techniques such as [BBL+10], capable of reusing invariants that have already been computed on subparts of the model. The incremental approach is particularly useful when multiple instances of the same components (atomic or composite) are used in the system. In such cases, it allows to factorize some part of the analysis. Thus, local invariants established on some part of the system can be automatically lifted to all similar parts within the system.

DFINDER originally implements efficient symbolic techniques for computing Boolean invariants ψ of the interactions between components [BGL+11]. It relies on the external third party tools Omega for constraints manipulation, Yices for SAT solving, and JavaBDD for BDDs. As shown in Figure 5, ψ can then be combined with the invariant ϕ_i of each constituent component to deduce a global invariant for the complete system (see [BLN+10] for a proof). At the same time, the tool also computes all the potential deadlock states denoted by DIS. If the formula $\wedge_i \phi_i \wedge \Psi \wedge DIS$ is unsatifiable, then the system is deadlock free. In the other case, the solutions denote some suspicious counter examples that can be reused by the tool to automatically refine the analysis. For the purpose of this work, we have implemented new techniques based on linear invariants in order to compute ψ.

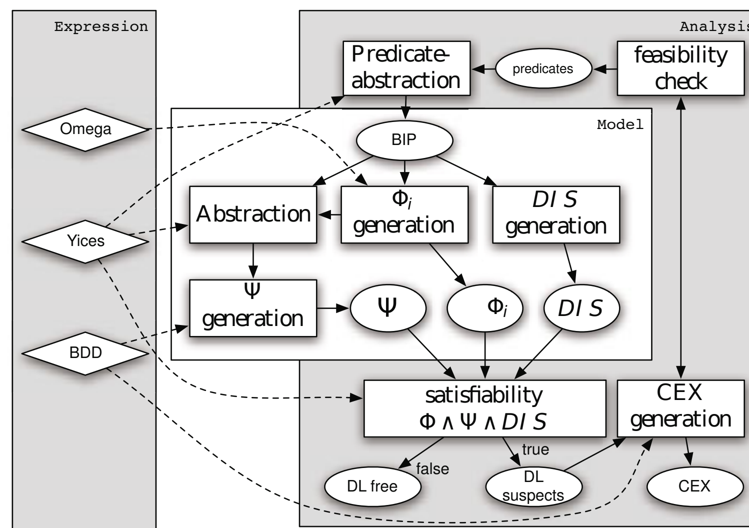

Figure 5: Structure of the D-Finder tool.

Until now, the DFINDER approach has been limited to invariants that can be represented by boolean formulas. This has been shown to be convenient in many contexts, going from simple to complex case studies [BSG+11]. However, there are situations where boolean invariants may not be appropriate. Consider the state variable at_l_i which monitors that some process is currently in control state l_i. Whatever the transition relation of the system is, DFINDER will only be able to generate invariants of the form e.g., $at_l_1 \vee at_l_2 \vee at_l_3$. Such an invariant ensures that (at least) one of the processes is in one of the control states l_1, l_2, and l_3, which is sometimes sufficient to infer the deadlock freeness. However, such invariants are not precise enough to prove a mutual exclusion property. Here, an invariant of the form $at_l_1 + at_l_2 + at_l_3 \leq 1$ would be needed, which shows that at most one

process can be in a critical state at any time. To reason on such more complex properties, we have to work with invariants capable of *counting* how many processes are at given states. A way to do this is to use linear invariants, i.e., invariants that can be represented by sets of linear equations. Such invariants have already been studied for a wide range of models for concurrent systems, and in particular for Petri Nets [UoTiC]. We propose new methods for linear invariant generations in BIP. More details about this contribution can be found in [BBBL12].

A BIP model is a set of interacting *components* $B_i = (L_i, P_i, \mathcal{T}_i)$, $1 \leq i \leq n$, such that each component B_i is a transition system defined by the transition relation $\mathcal{T}_i$ between control states L_i, labelled by ports P_i. An *interaction* a for components $\{B_i\}_{1 \leq i \leq n}$ is a subset of components ports such that $|a \cap P_i| \leq 1$ for all i, that is, at most one port of each component is involved in a. It defines a synchronization between involved components, that is, components B_i such that $|a \cap P_i| = 1$. Interaction a may take place if all the involved components B_i enable a transition labelled by $p_i = a \cap P_i$ from the current state. Building the composition $B = \gamma(B_1, \ldots, B_n)$ of components B_i with respect to a set of interactions γ faces the problem of state explosion for large systems, since its control states is given by the product $\mathcal{L} = L_1 \times \ldots \times L_n$. We consider *location variables* at_l, $l \in L = \cup_{1 \leq i \leq n} L_i$. Any global state $\ell \in \mathcal{L}$ of B is represented by an assignment of the location variables in which $at_l_i = 1$ if component B_i is in control location l_i, $at_l_i = 0$ otherwise. A linear equation for the composition $B = \gamma(B_1, \ldots, B_n)$ is a formula of the form

$$\sum_{l \in L} u_l \cdot at_l = u_0,$$

where $u_0, u_l \in \mathbb{Z}$. Such a constraint can be also represented by $\mathbf{u}^T \cdot At = u_0$, where $\mathbf{u} = (u_l)_{l \in L}$ and $At = (at_l)_{l \in L}$. A *linear invariant* for B is a constraint $\mathbf{u}^T \cdot At = u_0$ that holds in all the reachable states of B. Notice that given coefficients $\mathbf{u}$, u_0 is fully determined by the initial state of B, that is: $u_0 = \mathbf{u}^T \cdot At_0$, where At_0 is the vector of the location variables corresponding to the initial state.

To compute linear invariants for a composition $B = \gamma(B_1, \ldots, B_n)$, we represent the global system B as a Petri Net PN, in which places corresponds to control states of components B_i, and transitions corresponds to interactions γ. Linear invariants $\mathbf{u}^T \cdot At = u_0$ are solutions of a system of linear equations induced by the incidence matrix C of PN [Mur89], that is, they satisfy

$$C^T \mathbf{u} = \mathbf{0}. \tag{1}$$

To efficiently solve (1), we developed an algorithm based on a variant of Gauss-Jordan elimination that exploits the properties of the matrices we obtain from BIP systems (i.e. its particular structure and its sparsity). In addition to efficiency, one of the major advantages of our algorithm is that can be exploited to derive an incremental version [BBBL12].

We also implemented the proposed method in DFINDER and evaluated on several case studies [BBBL12]. The experiments show how our approaches outperform classical techniques on a wide range of models. Particularly, our method is as efficient as the one to compute boolean invariants, and it allows for finer state-space approximations (hence removing more spurious counter-examples), as shown in Figure 6. Finally, our results asses that DFINDER is faster than tools implementing classical mathematical approaches Gauss-Jordan (GAUSS) and the general Petri-net analyzer CHARLIE) (see Figure 7).

4 Advancements on Security Policies and Access Control

Security properties are essential for the kind of applications we consider in ASCENS. For instance, it is essential for the science cloud case study to be able to ensure privacy of the data as well as

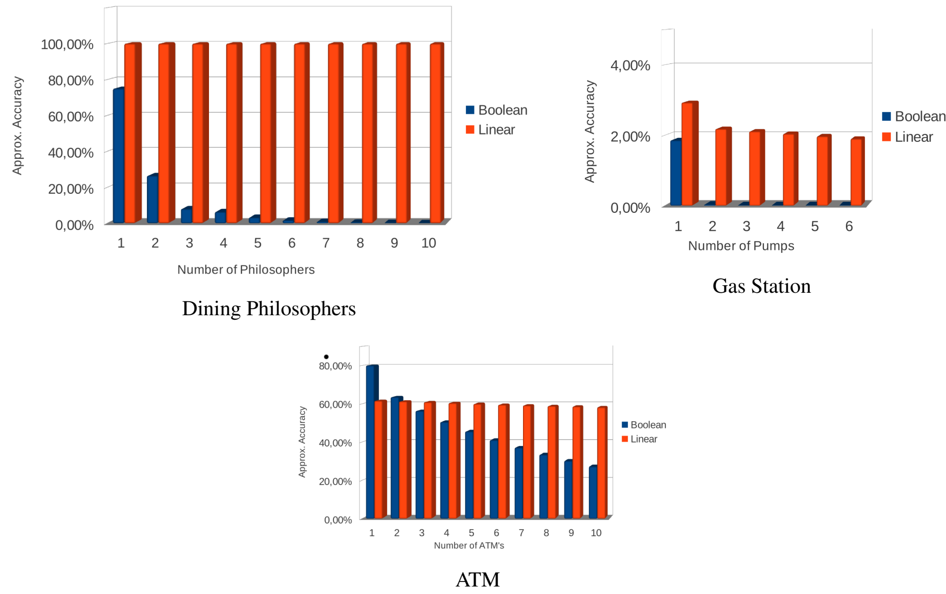

Figure 6: Preciseness of linear invariants w.r.t. boolean invariants.

Component information			Time (`m'ss`)			Memory (`Bytes`)		
scale	locations	interactions	CHARLIE	GAUSS	DFINDER	CHARLIE	GAUSS	DFINDER
DINING PHILOSOPHERS								
500 philos	3000	2500	8'40	0'03	>0'01	143M	120M	0.9M
1000 philos	6000	5000	74'42	0'13	0'01	468M	596M	1.0M
2000 philos	12000	10000	-	0'73	0'04	-	2.4G	1.2M
6000 philos	36000	30000	-	-	1'40	-	-	1.8M
9000 philos	54000	45000	-	-	9'15	-	-	2.0M
ATM								
50 machines	1812	1656	-	0'14	>0'01	-	73M	1.6M
100 machines	3612	3306	-	1'38	0'01	-	238M	2.8M
200 machines	7212	6606	-	12'41	0'03	-	940M	4.0M
400 machines	14412	13206	-	-	0'13	-	3.6G	6.4M
500 machines	18012	16506	-	-	0'31	-	-	7.2M
GAS STATION								
50 pumps	2152	2000	-	1'17	0'01	-	69M	2.5M
100 pumps	4302	4000	-	14'58	0'04	-	271M	3.3M
200 pumps	8602	8000	-	-	0'14	-	-	4.7M
500 pumps	21502	20000	-	-	2'30	-	-	8.7M
700 pumps	30102	28000	-	-	3'40	-	-	11.4M
READERS - WRITERS								
50 writers	1152	1650	3'15	1'06	>0'01	150M	54M	2.2M
100 writers	2322	3300	19'50	8'12	0'02	937M	212M	2.6M
200 writers	4642	6600	-	65'43	0'06	-	847M	3.2M
500 writers	11502	16500	-	-	0'37	-	-	5.0M
1000 writers	23002	33000	-	-	3'22	-	-	7.5M
2000 writers	46002	66000	-	-	17'40	-	-	9.7M
SMOKERS								
300 smokers	906	901	0'17	0'30	0'01	90M	14M	1.4M
600 smokers	1806	1801	1'31	3'11	0'01	229M	52M	2.3M
1500 smokers	4506	4501	-	55'00	0'06	395M	319M	3.1M
6000 smokers	18006	18001	-	-	1'51	-	-	6.8M
9000 smokers	27006	27001	-	-	4'37	-	-	9.3M

Figure 7: Execution time and memory footprint for some examples.

availability of the services offered by the cloud, even if the system encounters malicious attacks. Proving or ensuring security properties requires dedicated methods and tools. The methods presented in Sections 1 and 2 target usual and general properties for systems, such as safety (e.g. deadlock freedom, mutual exclusion, reachability of non-safe states) and liveness, and thus are not efficient for checking security properties. The contributions presented below specialize the design flow considered in ASCENS for access control (Section 4.1) and availability of service properties (Sections 4.2).

4.1 Model-Driven Development of Access Control Policies

Access control is a fundamental means for restricting what kind of operations (authenticated) users can perform on protected resources. Different access control systems have been developed to support security. The main components of these systems are the security policies, the security model, and the implementation mechanisms [DSJ05]. The policies define the rules according to which access control must be regulated. The model provides a formal representation of the policies and how they work and finally, the mechanisms define how the controls imposed by the policies and the model are implemented.

Languages for access control aim at supporting the expression and enforcement of policies. Several such languages have been proposed. Many of them are XML-based, e.g., the OASIS standard eXtensible Access Markup Language (XACML) [OAS05] providing features to express authorization rules. However, the XML syntax of XACML can make the task of writing policies difficult and error-prone, and it is not adequate for a formal definition of the semantics of the language and reasoning. Other languages rely on concepts and techniques from logic, which instead offer the advantage of a formal foundation and a well-defined analysis. But these formal languages have the drawback of not being usable for a wide spectrum of users.

Our contribution is to make the specification of access control policies accessible to people not necessarily familiar with formal languages. The solution we illustrate hereafter is to initially specify the security requirements using a high-level, graphical, modeling language (called UWE [KKZB08]), thus also providing a human understandable view of the policies in force at the system, and then to automate the policy development process towards a formally founded language (called FACPL [MPT12]) by means of model-driven approach, implemented as a software tool-chain.

The approach offers the advantages of an easy to learn and intuitive visual specification language for policies and a formal specification in the background enabling evaluation of policies and access requests through a FACPL Policy Decision Point [BKM$^+$12]. This model-driven process is shown in Figure 8. It solves the problems mentioned above as the translation from one specification into the other can performed automatically. In practice, it is implemented as a transformation in two steps using XACML as intermediate language. Thus the tool-chain comprises two transformations: UWE2XACML and XACML2FACPL[1].

Modeling with UWE. The UML-based Web Engineering (UWE) [KKZB08] is an engineering approach for modeling web applications. UWE uses the extension mechanisms provided by UML via the definition of a UML profile, which provides a set of stereotypes, tag definitions and constraints. One of the cornerstones of the UWE language is the "separation of concerns" principle using separate models for views such as content, navigation, presentation, processes, etc. The most relevant UWE models for this work are: (1) The *Content Model*, representing the domain concepts that are relevant for the web application and the relationships between them. (2) The *Role Model*, defining a hierarchy of user groups with the purpose of authorization and access control. It is usually included

[1]Instructions and software for installing the tool-chain can be found at `http://uwe.pst.ifi.lmu.de/uwe2facpl`

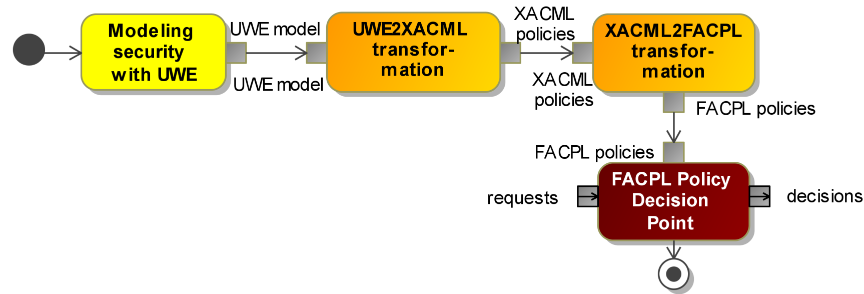

Figure 8: Toolchain for the model-driven approach.

in a *User Model*, which specifies basic structures such as, e.g. that a user can take on certain roles simultaneously. (3) The *Basic Rights Model*, expressing role based access control on the domain concepts specified in the content model, picking the roles from a user model. (4) The *Navigation Model*, providing a graphical representation of the path the user can navigate in the web system. In this model also security features are represented as, e.g., authentication, access control and secure connections [BKK11]. This model also represents security features as, e.g., authentication, access control and secure connections [BKK11].

For each view, an appropriate type of UML diagram is selected and a set of stereotypes, tag definitions and constraints is provided. Concepts of the content and role models and their relationships are shown as classes and associations in a UML class diagram. The basic rights model connects role instances with content classes or their attributes/methods using stereotyped dependencies. These dependencies specify create/read/update/delete/execution rights. For the navigation model, UWE provides two graphical representations: a structural visualization as UML stereotyped class diagrams and a behavioral form using UML state machines.

Transforming UWE models to XACML. Intuitively, the UWE2XACML transformation generates a `<PolicySet>` for each role, each of which contains one `<Policy>` for any class connected to the considered role. Furthermore, a single `<Policy>` is used to deny access to all resources not specified in the `<PolicySet>`, which is the default behavior of UWE's basic rights models.

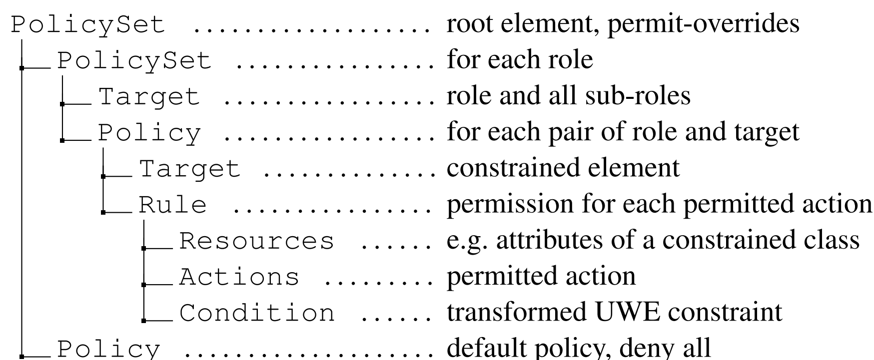

Figure 9: Model transformations to generate XACML policies.

Notably, to allow a sub-role of a given role to use the permission specified by the super-role, the target of the `<PolicySet>` corresponding to the super-role is extended to also match requests from the sub-role (e.g. the target of the `<PolicySet>` for `receptionist` specifies two subjects, with

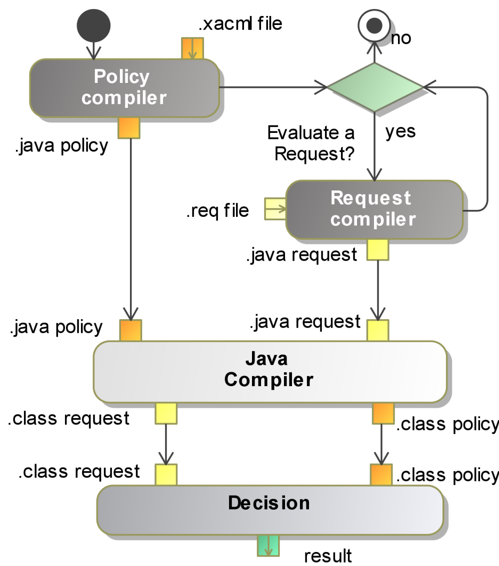

Figure 10: FACPL Policy Decision Point.

roles `receptionist` and `physician`, respectively).

Each `<Policy>` for a constrained class contains one `<Rule>` for each action between the role and the class. Attributes targeted by `*All` actions are divided into a set of `<Resources>`, omitting those from the except tag. OCL constraints inside UML comments with authorizationConstraint stereotype are transformed to a `<Condition>`. The condition is located within a `<Rule>` representing the appropriate action. For the time being, we implemented only a few basic OCL constraints.

Translating policies from XACML to FACPL. The transformation, performed by the XACML2 FACPL component in Figure 8, is straightforward. Its flow loops over the policy sets creating the necessary data structures for the FACPL representation. The original XML document is read by using JAXB[2]. The loop over the elements is driven by the XACML schema definitions by traversing its data types.

Evaluating policies with FACPL Policy Decision Point. The implementation[3] of the FACPL language is made in Java. The workflow of such a tool is graphically depicted in Figure 10. This tool "compiles" a policy written in the FACPL syntax into a Java class following the semantics rules defined in [MPT12]. Thus, a repository storing some policies, and the related PDP, consists of a Java archive containing all the Java classes generated from the policies. Similarly, an access request is compiled into a Java class. A policy decision is then computed by executing the generated policy code with the request code passed as parameter to an entry method. The generated PDP can be then integrated as a module into the code of the main web application, possibly obtained from other UWE models.

For further details on this work we refer the interested reader to [MPT12] for the FACPL semantics and to the publication [BKM$^+$12] for the full description of the tool-chain.

[2]JAXB. `http://jaxb.java.net`

[3]Source and binary code of the FACPL implementation are available from `http://rap.dsi.unifi.it/xacml_tools`

4.2 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.

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 constant quantity, which does not change over time, regardless of how bad the DoS attack can get. Cloud Computing, by offering the possibility of dynamic resource allocation, can be used to leverage stable availability when combined with DoS defense mechanisms. Service-oriented systems such as the MasterCard service are distributed systems operating in a dynamically changing environment. They need to cope with changing numbers of user demands and with hostile attacks. To be used/operated safely, services have to satisfy functional as well as non-functional requirements and it is not a priori clear what is the best realization of a service in each particular situation. Model-driven approaches to service development offer the possibility of tackling these issues at a high level of abstraction during early stages of system analysis and design. In particular, design patterns have been successfully used for improving programming solutions in several domains, including object-orientation [GHJV95], service-oriented computing [M. 08, Erl08] and security [SFBH$^+$05]. Patterns are general, reusable solutions to commonly occurring problems in software design; they clearly define the programming context, the problem and the advantages and disadvantages of design solutions (see e.g., [GHJV95, SFBH$^+$05]).

In [EMA$^+$12a] we considered a formal pattern-based approach to the design and mathematical analysis of security mechanisms of Cloud services. We have shown that formal patterns can help to deal with security issues and that formal analysis can help to evaluate patterns in various contexts. We specified the dynamic Server Replication (SR) and the Adaptive Selective Verification (ASV) protocols as formal patterns in the executable rewriting logic language Maude. By formally composing the two patterns we have obtained the new pattern ASV$^+$SR. We have analyzed properties of the ASV+SR pattern using the statistical model checker PVeSta, and were able to show as our main result that, unlike the two original patterns, ASV$^+$SR achieves stable availability in presence of a large number of attackers at reasonable cost, which can be predictably controlled by the choice of the overloading parameter.

Our current results rely on two simplifications: The client-server communication consists of a stateless request-reply interaction and the replication of servers is only able to add but not to delete servers. As next steps, we plan to refine the patterns to cope with the winding-down of resources at the end of a DoS attack and with more complex client-server interactions where the server has to preserve state. Moreover, in this work we have only studied quantitative properties of the patterns; it would be very interesting and useful to analyze also qualitative properties. In [CGM$^+$08] it is shown that adding cookies to a client-server system preserves all safety properties. We conjecture that the same holds for the ASV and ASV$^+$SR protocols. Finally, we plan to continue with our pattern-based approach and to build a collection of formal patterns for security mechanisms. For further details on this work we refer the interested reader to [EMA$^+$12b].

5 Advancements on Verification of SC Implementation Compliance with High-level Specification

Methods developped in Tasks T5.1, T5.2 and T5.3 are all based on high-level models of Service Components (SCs) and Service Component Ensembles (SCEs). By keeping high level information, they take advantage of the special characteristics of SCEs allowing to master their complexity. However, valid models may still lead to incorrect systems if the low-level of code implementing the SCs is not compliant with their high-level behavior specifications.

In Task T5.4 we provide means for reasoning about the "business code" of an SC, allowing to check that it is correctly implemented by the low-level code. This makes an important contribution to the SCs development cycle since the business code of an SC is hand-written (i.e. provided by the developer), thus it is prone to be inconsistent with its specification (either when created or when changes occur either in the code or in the high-level specification as a consequence of emerging requirements). The task consists of two parts, divided according to implementation language of SCs. First, it addresses the SCs implemented in the C and C++ languages; second, it focuses on SCs in Java. Although the basic approach and the verification task are very similar in both cases, the technical differences between the languages do not allow to use a single tool in an efficient way. Therefore, two software tools will be incorporated.

5.1 GMC

For verification of the C/C++ SCs, the GMC model checker developed formerly at CUNI is planned to be used. During the second project year, we have worked on an extension of the tool to support the C++ language, which was not available before. After the implementation of GMC is completed, the plan is to apply it on the ARGoS case study to verify several properties either encoded as asserts in the code, or specified externally.

GMC is connected with the GCC compiler [GCC] in the sense that it uses the GCC's intermediate code from the LTO (link-time optimization) phase as code representation. This intermediate code, called GIMPLE, is created by GCC and saved to a file. Then, either simulation or verification can take place upon this representation. The advantage of this approach is first that the GIMPLE code can be already optimized and thus closer to what is really executed. Second, taking into account the complexity of the C language specification, since the GIMPLE code is much simpler that the original source code, GMC follows the semantics chosen (in cases of ambiguities) by GCC.

As to the motivation of development of "yet another model checker", we are not aware of any other code model checking tool that would combine the set of features available in GMC in a single tool. First, GMC implements a general memory model thus supporting dynamic allocations on the heap, second, it supports multithreaded programs. Here, since there is no threading standard for C/C++ languages, a relatively generic threading support was implemented and so far, it was instantiated for PThreads [pth].

5.2 JPF

For verification of the SCs in Java, Java PathFinder [JPF] is employed. Based on our previous work [PPK06], the verification framework will be extended to handle the specifics of SC implementation, especially in the sense of open code (with missing dependencies, main method, etc...). This will make it possible to apply it on the code of particular SCs, making the verification modular and thus scalable. We plan to demonstrate the functionality of the framework via application on the jDEECo framework and applications built upon it. In this project year, we make the implementation

of jDEECo, the implementation/level extension of SCEL, compliant with the set of features supported by JPF, which is not complete (especially in the sense of standard libraries, generics combined with reflection, etc.).

5.3 Development of methods

We have also investigated options for improving the methods of program model checking in general, at the theoretical level; once finished, the improved techniques will be implemented in one of the aforementioned cases. One of the most important issues of program model checking is its scaling. Therefore, it is desirable to face the problem of state-space explosion to make the technique scale to real-life applications. In this sense, our research in this year has been focused on the partial-order-reduction optimization. Further pruning the state space of the program can be achieved via deduction of ownership of data objects. If an object has a unique owner (at a particular program point), it means that it cannot be modified by another thread. Knowing this, the number of states to be explored can be significantly decreased even if no explicit locking is present in the implementation – if it is proved that an object cannot be simultaneously modified, the instructions accessing it need not to be considered as scheduling-relevant.

6 Conclusion and Summary of the Main Achievements

During the second year, we have worked in the following research directions incorporating quantitative aspects of a SC or SCE:

- We presented an example motivating the need for a new definition of program repair. We defined a new notion of repair for reactive programs and presented an algorithm to compute such repairs. The algorithm is based on synthesizing repairs with respect to a lower and an upper bound on the set of generated traces. We showed the limitations of any repair approach that is based on preserving part of the program's behavior.

- We provided the first semi-symbolic algorithm and implementation to find optimal strategies for MDPs with ratio objectives (Ratio-MDPs). Our approach avoids the need for unichain MDPs, which was a major bottleneck in the implementation of [Bry92]. Our implementation is fully integrated into the PRISM model checker, providing easy- to-use tool support for the ratio optimization criterion. We also added human-readable output of strategies to PRISM. Our work is related to the work of Wimmer et al. [WBB+10], which performs semi-symbolic computation for MDPs with average objectives (a special case of the ratio objectives). We show that our implementation can also outperform theirs.

- We improved the existing compositional verification technique for BIP programs, implemented by the tool D-Finder. We introduced linear invariants in addition to boolean invariants already considered in previous verification techniques. Our methods build on transitions of components that are abstracted by linear equations and then combined to form a system of equations. We show that each solution of such system is a linear invariant. To achived scalability, we propose an online algorithm that processes equations in the system in an iterative manner based on the structure of the underlying component-based system. As a second contribution, we proposed an incremental extension of the proposed approach. Our new contributions have been implemented in D-finder and evaluated on several case studies. The experiments show how our approaches outperform classical techniques on a wide range of models.

- In this project year, we have worked on extension of GMC with support of C++, which is to be completed in the near future. Also, we have created a local version of the jDEECo framework and adapt it to be compliant to the input language of JPF and discovered the potential practical issues with model checking it. As to the methods, we have worked on techniques for deducing ownership of objects; this information can be used for making partial order reduction more powerful, i.e., to handle larger state spaces.

7 Next Step and Long-Term Technical goals

For program repair, we will follow two orthogonal directions to make it possible to repair more machines. In the first one we will try to know as soon as possible when a machine will fail or succeed. For this, studying repairs with finite look-ahead is an interesting direction. The second direction studies a relaxed notion of set-inclusion in order to express how “close” two machines are. For instance, it is possible to measure the distance between a repair and a machine quantitatively, e.g,. by measuring how often they differ on average, or when the last point in time is where a different output occurs.

For computation of controllers in probabilistic environments, we envision future work to go into the following three areas.

- First, we want to apply the current algorithms on bigger case studies. We believe that probabilistic environments and efficiency is of interest to industry. We will therefore look for opportunities that allow this strong combination to have impact outside of academia.
- The implementation we currently have is research software, and as such of sub-optimal efficiency. Future work could therefore investigate the two bottlenecks of the current implementation: end-component decomposition and lumping.
- In [vEJ12a] Jobstman et al. looked at the application of the ratio objective to two-player games. They looked at 1 1/2 player games (MDP), but 2 1/2 player games (i.e. games with two players and probabilities), remain unstudied.

Regarding access control policies we intent to integrate tools for analyzing policies and generating test cases, such as Margrave [FKMT05] and X-CREATE [BDLM12], respectively. Our formal access control policy language (FACPL) is the basis for the former; the intermediate transformation to XACML will enable the latter.

In the next year, we also plan to finish the support for C++ in GMC and to apply the tool on the Argos case study. As for the jDEECo, we plan to verify interesting properties of the framework and later also properties of applications built upon jDEECo. Also, once the research is finished, we plan to implement the ownership deduction to JPF and demonstrate its usefulness.

References

[BBBL12] Saddek Bensalem, Benoit Boyer, Marius Bozga, and Axel Legay. Incremental generation of linear invariants for component-based systems. Technical Report TR-2012-15, Verimag Research Report, 2012. `http://www-verimag.imag.fr/TR/TR-2012-15.pdf`.

[BBL+10] S. Bensalem, M. Bozga, A. Legay, Th.H. Nguyen, J. Sifakis, and R. Yan. Incremental component-based construction and verification using invariants. In *FMCAD’10*, 2010.

[BBNS08] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis. Compositional verification for component-based systems and application. In *ATVA*, 2008.

[BBNS09] S. Bensalem, M. Bozga, T-H. Nguyen, and J. Sifakis. D-Finder: A tool for compositional deadlock detection and verification. In *CAV*, LNCS. Springer, 2009.

[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.

[BEGL99] Francesco Buccafurri, Thomas Eiter, Georg Gottlob, and Nicola Leone. Enhancing model checking in verification by ai techniques. *Artif. Intell.*, 112(1-2):57–104, 1999.

[BGL+11] S. Bensalem, A. Griesmayer, A. Legay, T.H. Nguyen, J. Sifakis, and R. Yan. D-finder 2: Towards efficient correctness of incremental design. In *NASA Formal Methods*, LNCS. Springer, 2011.

[BIP] BIP – incremental component-based construction of real-time systems. `www.bip-components.com`.

[BKK11] Marianne Busch, Alexander Knapp, and Nora Koch. Modeling Secure Navigation in Web Information Systems. In *BIR*, LNBIP 90, pages 239–253. Springer, 2011.

[BKM+12] Marianne Busch, Nora Koch, Massimiliano Masi, Rosario Pugliese, and Francesco Tiezzi. Towards Model-Driven Development of Access Control Policies for Web Applications. In *Workshops at MoDELS 2012*. ACM DL, 2012. to appear.

[BLN+10] S. Bensalem, A. Legay, T.H. Nguyen, J. Sifakis, and R. Yan. Incremental invariant generation for compositional design. In *TASE*, 2010.

[BNR03] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *POPL 2003*, pages 97–105, January 2003.

[Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[BSG+11] S. Bensalem, L. Silva, A. Griesmayer, F. Ingrand, A. Legay, and R. Yan. A formal approach for incremental construction with an application to autonomous robotic systems. In *SC'11*, LNCS. Springer, 2011.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs*, 1981.

[CGM+08] Rohit Chadha, Carl A. Gunter, José Meseguer, Ravinder Shankesi, and Mahesh Viswanathan. Modular Preservation of Safety Properties by Cookie-Based DoS-Protection Wrappers. In *FMOODS*, volume 5051 of *LNCS*, pages 39–58, 2008.

[CGMZ95] E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *DAC*, pages 427–432, San Francisco, CA, June 1995.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model checking*. MIT Press, 1999.

[Chu63] A. Church. Logic, arithmetic and automata. In *Proc. 1962 Int. Congr. Math.*, pages 23–25, 1963.

[CMB08] Kai-Hui Chang, Igor L. Markov, and Valeria Bertacco. Fixing design errors with counterexamples and resynthesis. *IEEE Trans. on CAD*, 27(1):184–188, 2008.

[CTBB11] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *ICSE 2011*, pages 121–130, New York, NY, USA, 2011. ACM.

[dA97] L. de Alfaro. *Formal Verification of Probabilistic Systems*. PhD thesis, Stanford University, 1997.

[DSJ05] S. De Capitani di Vimercati, P. Samarati, and S. Jajodia. Policies, models, and languages for access control. In *DNIS*, LNCS 3433, pages 225–237. Springer, 2005.

[EKB05] Ali Ebnenasir, Sandeep S. Kulkarni, and Borzoo Bonakdarpour. Revising unity programs: Possibilities and limitations. In *OPODIS*, 2005.

[ELLL01] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Trail-directed model checking. *ENTCS*, 5(3), August 2001. Software Model Checking Workshop 2001.

[EMA+12a] Jonas Eckhardt, Tobias Mühlbauer, Musab AlTurki, José Meseguer, and Martin Wirsing. Stable availability under denial of service attacks through formal patterns. In *Proceedings of the 15th international conference on Fundamental Approaches to Software Engineering*, FASE'12, pages 78–93, Berlin, Heidelberg, 2012. Springer-Verlag.

[EMA+12b] Jonas Eckhardt, Tobias Mühlbauer, Musab AlTurki, José Meseguer, and Martin Wirsing. Stable availability under denial of service attacks through formal patterns. In *FASE*, pages 78–93, 2012.

[Erl08] Thomas Erl. *SOA Design Patterns*. Prentice Hall, 2008.

[FKMT05] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In *ICSE*, pages 196–205. ACM, 2005.

[FKN+11a] V. Forejt, M. Kwiatkowska, G. Norman, D. Parker, and H. Qu. Quantitative multi-objective verification for probabilistic systems. In *Proc. TACAS'11*, 2011.

[FKN+11b] Vojtech Forejt, Marta Z. Kwiatkowska, Gethin Norman, David Parker, and Hongyang Qu. Quantitative multi-objective verification for probabilistic systems. In *TACAS*, pages 112–127, 2011.

[FMY97] M. Fujita, P. C. Mcgeer, and J. C. Y. Yang. Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation. *Formal Methods in System Design*, V10(2):149–169, April 1997.

[GBC06] Andreas Griesmayer, Roderick Bloem, and Byron Cook. Repair of boolean programs with an application to c. In *CAV*, volume 4144 of *LNCS*, pages 358–371. Springer, 2006.

[GCC] GNU Compiler Collection.
`http://gcc.gnu.org/`.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GV03] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *10th International SPIN Workshop*, pages 121–135. Springer-Verlag, 2003. LNCS 2648.

[JGB05] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In *CAV*, pages 226–238, 2005.

[JM06] M. U. Janjua and A. Mycroft. Automatic correction to safety violations in programs. Thread Verification (TV'06), 2006. Unpublished.

[JPF] Java PathFinder. `http://babelfish.arc.nasa.gov/trac/jpf/`.

[JPJ82] J-P.Queille and J.Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, LNCS. Springer, 1982.

[JRS02] H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. In *TACAS'02*, pages 445–459, Grenoble, France, April 2002. LNCS 2280.

[JSGB12] Barbara Jobstmann, Stefan Staber, Andreas Griesmayer, and Roderick Bloem. Finding and fixing faults. *Journal of Computer and System Sciences (JCSS)*, 78(2):441–460, 2012.

[KKZB08] Nora Koch, Alexander Knapp, Gefei Zhang, and Hubert Baumeister. UML-Based Web Engineering - An Approach Based on Standards. In *Web Engineering*, Human-Computer Interaction Series, pages 157–191. Springer, 2008.

[KNP11] M. Z. Kwiatkowska, G. Norman, and D. Parker. Prism 4.0: Verification of probabilistic real-time systems. In *CAV*, pages 585–591, 2011.

[LR81] D. J. Lehmann and M. O. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *POPL*, 1981.

[M. 08] M. Wirsing et al. Sensoria Patterns: Augmenting Service Engineering. In *ISoLA*, volume 17 of *CCIS*, pages 170–190, 2008.

[Mas] MasterCard. MasterCard Statement. `http://www.businesswire.com/news/home/20101208005866/en/MasterCard-Statement` (09/2011).

[MPT12] Massimiliano Masi, Rosario Pugliese, and Francesco Tiezzi. Formalisation and Implementation of the XACML Access Control Mechanism. In *ESSoS*, LNCS 7159, pages 60–74. Springer, 2012.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *IEEE*, 1989.

[NPK+05] Gethin Norman, David Parker, Marta Z. Kwiatkowska, Sandeep K. Shukla, and Rajesh Gupta. Using probabilistic model checking for dynamic power management. *Formal Asp. Comput.*, 17(2):160–176, 2005.

[OAS05] OASIS XACML TC. eXtensible Access Control Markup Language (XACML) version 2.0, 2005.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS*, 1977.

[PPK06] Pavel Parizek, Frantisek Plasil, and Jan Kofron. Model Checking of Software Components: Combining Java PathFinder and Behavior Protocol Model Checker. In *Proceedings of 30th Annual IEEE/NASA Software Engineering Workshop SEW-30 (SEW'06)*, pages 133–141. IEEE Computer Society, 2006.

[pth] POSIX Threads. `http://en.wikipedia.org/wiki/POSIX_Threads`.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, pages 337–351, 1982.

[RR03] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *ICASE*, pages 30–39, Montreal, Canada, October 2003.

[RS04] K. Ravi and F. Somenzi. Minimal assignments for bounded model checking. In *TACAS'04*, pages 31–45, Barcelona, Spain, March-April 2004. LNCS 2988.

[SDE08] Roopsha Samanta, Jyotirmoy V. Deshmukh, and E. Allen Emerson. Automatic generation of local repairs for boolean programs. In *FMCAD*, pages 1–10, 2008.

[SFBH+05] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. *Security Patterns*. Wiley, 2005.

[UoTiC] University of Technology in Cottbus. Charlie: a tool to analyse Petri nets. `http://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Charlie`.

[vEJ12a] C. von Essen and B. Jobstmann. Synthesizing efficient controllers. In *VMCAI*, pages 428–444, 2012.

[vEJ12b] Christian von Essen and Barbara Jobstmann. Program repair revisited. Technical Report TR-2012-4, Verimag Research Report, 2012. `www-verimag.imag.fr/TR/TR-2012-4.pdf`.

[VYY09] Martin Vechev, Eran Yahav, and Greta Yorsh. Inferring synchronization under limited observability. In *TACAS'09*, volume 5505 of *LNCS*, pages 139–154. Springer, 2009.

[WBB+10] R. Wimmer, B. Braitling, B. Becker, E. M. Hahn, P. Crouzen, H. Hermanns, A. Dhama, and O. Theel. Symblicit calculation of long-run averages for concurrent probabilistic systems. In *QEST*, 2010.

[YBK10] H. Yue, H. C. Bohnenkamp, and J.-P. Katoen. Analyzing energy consumption in a gossiping mac protocol. In *MMB&DFT 2010*, pages 107–119, 2010.

[ZH02] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.