Online Failure Prediction of Dynamically Evolving Systems

Maria Rita Di Berardini, Henry Muccini, Andrea Polini, Pengcheng Zhang

Abstract—When dealing with dynamically evolving systems, components may be inserted or removed while the system is being operated. Unsafe run-time changes may compromise the correct execution of the entire system, and online analysis techniques have been proposed to mitigate the effects of such failures. With CASSANDRA we want to move one step ahead, by defining a novel proactive monitoring and verification technique with the ability to predict (and prevent) the potential failures happening in the future. CASSANDRA combines design-time and run-time information for implementing an online failure prediction approach: run-time information is used to identify the current execution state, and to drive the construction of a design-time model that looks $k$ steps ahead of the current execution state. Such a model is used to check whether a set of desired properties will be potentially neglected in near future executions. CASSANDRA algorithms have been formally defined and the approach has been concretized into the OSGi Platform, applied to a realistic case study and evaluated.

I. INTRODUCTION

In the last years software production has been continuously moving toward dynamic systems like SOA-based and the Future Internet-oriented applications [1]. Preventing failures of those dynamically evolving systems is far from simple: new components or services\(^1\) may come from partially unknown third parties, thus limiting the trust on their correct and unmalicious behavior; system requirements or contextual situations may require the dynamic integration of new components, even if they can potentially lead to erroneous states.

In such a context, traditional off-line integration and regression testing strategies may not be enough and have to be complemented with run-time validation and verification activities. The number of possible configurations for a dynamically evolving system may become unmanageable (see e.g. [2]); so it may become impossible to validate all of them. Even the off-line analysis of some configurations may be inefficient, since the lifetime of a certain configuration can be limited with a small fragment of the provided behavior exercised before the system evolves. In this scenario, new verification techniques need to be devised, that may take advantage of both (classical) design-time and (more recent) run-time verification techniques, that can efficiently and effectively manage frequent and run-time updates, and may enable actions to predict and prevent potential failures.

Online failure prediction consists in forecasting, while the system is running, failure occurrences in the near future, by using the current state of a system, and frequently, the past experience [3]. Unlike other predicting methods, online failure prediction is based on short-term assessment (predicting the near term future is, in principle, more successful than attempting long term predictions) and made on the basis of the current system state (typically identified through runtime monitoring techniques). As remarked in [3], dependability and resilience are and will remain a permanent challenge due to (among others) the frequent configuration updates and upgrades in dynamic systems, and the use of third-party, Commercial-Of-The-Shelf components: for those systems, "... runtime monitoring and online failure prediction seem to be one of a very few alternatives for effective online dependability assessment and enhancement ....".

Goal of this research paper is to propose an online failure prediction technique for dynamically evolving systems\(^2\). Towards this research goal, we here propose CASSANDRA, a novel proactive monitoring [5] approach to predict potential failures by looking ahead the current execution state. To this end, CASSANDRA combines design-time and run-time information for proactive run-time verification of dynamic component-based systems. In short, CASSANDRA uses run-time information to identify the current execution state, and checks whether the projection of a design-time model $k$ steps ahead of the current state satisfies a set of wanted/unwanted properties. The execution state is captured by monitoring the component-based system execution. Each component

\(^1\)Hereafter, we will use the term component(s), to refer to both concepts.

\(^2\)Fault diagnosis [4] and prevention actions will be addressed in future work.
Definition 1: An interface automaton is a tuple $P = (V_P, V_P^{\text{init}}, A_P, A_P^O, A_P^H, T_P)$ where:
- $V_P$ is a set of states and $V_P^{\text{init}}$ is a set of initial states that contains at most one state.
- $A_P^I$, $A_P^O$ and $A_P^H$ are mutually disjoint sets of input, output and internal actions. We define $A_P^I = A_P^I \cup A_P^O \cup A_P^H$. $P$ is closed if it has only internal actions (i.e. if $A_P^O = A_P^H = \emptyset$); otherwise $P$ is open.
- $T_P \subseteq V_P \times A_P \times V_P$ is a set of steps.

An action $a \in A_P$ is enabled at a state $v \in V_P$ if there is a step $(v, a, v') \in T_P$. The sets $A_P^I(v)$, $A_P^O(v)$ and $A_P^H(v)$ are the subsets of input, output and internal actions that are enabled at $v$. $A_P(v) = A_P^I(v) \cup A_P^O(v) \cup A_P^H(v)$. Interface automata are not required to be input-enabled, i.e. we do not assume that $A_P^I(v) = A_P^I$ for each $v \in V_P$. The inputs in $A_P(v) \setminus A_P^I(v)$ are called illegal inputs at $v$. $T_P^I = \{ (v, a, v') \in T_P | a \in A_P^I \}$ is the set of input steps; similarly we define the sets $T_P^O$ and $T_P^H$ of output and internal steps. Finally, a state $u$ is reachable from $v$ if there is an execution, i.e. an alternating sequence of states and actions of the form $v = v_0, a_0, v_1, a_1, \ldots, v_n = u$ where each $(v_i, a_i, v_{i+1}) \in T_P$. }

Two interface automata are composable if their actions are disjoint, except that some input actions of one automaton can be output actions of the other one [6].

Definition 2: Two interface automata $P$ and $Q$ are composable if $A_P^I \cap A_Q = A_P^O \cap A_P = \emptyset$ and $A_P^I \cap A_Q^O = A_P^O \cap A_Q^I = \emptyset$. Note that, if $P$ and $Q$ are composable, then $\text{shared}(P, A) = A_P \cap A_Q = (A_P^I \cap A_Q^O) \cup (A_Q^O \cap A_P^I)$.

Essentially, $P$ and $Q$ are mutually composable whenever $\text{shared}(P, Q) = (A_P^I \cup A_Q^O) \cup (A_Q^I \cap A_P^O)$. If $P$ and $Q$ are composable, their product $P \otimes Q$ is an interface automaton (whose set of states is $V_P \times V_Q$) that it will synchronize on shared actions, while asynchronously interleave all other (i.e. internal) actions.

Since $P$ and $Q$ are not required to be input-enabled, their product may have one or more states where one component produces an output that the other one is not able to accept. The states where this happens, i.e. all pairs $(v, u) \in V_P \times V_Q$ where there is an action $a \in \text{shared}(P, Q)$ such that either $a \in A_P^I(v) \setminus A_Q^O(u)$ or $a \in A_Q^O(u) \setminus A_P^I(v)$, are called illegal [6]. In [6], $P$ and $Q$ can be used together if there is a legal environment, i.e. an environment that can prevent (by generating appropriate inputs) $P \otimes Q$ from entering its illegal states.

II. BACKGROUND MATERIAL ON INTERFACE AUTOMATA

Interface automata have been introduced in [6] as a lightweight formalism for modeling temporal aspects of software components interfaces. As the I/O automata introduced by Lynch et al. in [8], interface automata interact through the synchronization of input and output actions, and asynchronously interleave all the other (i.e. internal) actions. Below we provide some basic definition taken from [6].
The execution of one exploration step by the run-time checking algorithm is faster than the execution of the corresponding step by the real system. This assumption is necessary to keep the exploration synchronized with the real execution. The release of this assumption would require the introduction of specific mechanisms to keep the monitor synchronized with the running system.

The online failure prediction algorithm we have implemented (described in Section III-C) relies on a specification of the component behaviour based on the interface automata formalism. The algorithm suitably composes the specifications of those components under execution. Instead of using the classical composition operator defined in [6], we base our algorithm on a slightly different composition operator (defined in Section III-B, Def. 3) that, according to us, is more suitable for the online prediction of failures in a dynamic environment. For our composition operator, any couple of components sharing a set of I/O actions can be integrated. Then, the composed automata is navigated by looking ahead to the current execution state. It is the task of our failure prediction approach to check whether the system is approaching an illegal state, and so to inform a possible failure avoidance mechanism that will take care of repair actions. An illegal state corresponds to an integration failure, by any path shorter than \( k \) steps and originating in the current state. In a sense, we assume that an illegal state can be reached as consequence of a wrong invocation/message done by one component on a component that either is not willing to accept it in the current state or does not exist at all.

Figure 1 sketches the general idea of our algorithm in case of two simple components \( P \) and \( Q \). Component \( Q \) models a file in a file system; after its opening, it can be repeatedly read and written until its closure. The automaton \( P \) represents a process that wants to use the resource made available by \( Q \).

First, the specifications of the two executing components are composed. For example, when both components \( P \) and \( Q \) are in state 2, \( P \) can perform an open action; this is a shared output whose corresponding input at the moment is not available. As described above, this invocation causes the evolution of the composed system into the illegal state \((3, 2)\) (see the lower part of Figure 1).

Then, the failure prediction approach we present in this paper explores the composed model as illustrated in the lower part of Figure 1. Starting from the initial state \((1, 1)\), and assuming a look-ahead \( k = 2 \), the failure prediction approach explores the possible future states reachable by any path longer at most 2. The initial configuration (associated to state \((1,1)\)) is described in step 1 of Figure 1. After the open, the failure prediction mechanism will move the explored model of a further step assuming as current state the pair \((2, 2)\) and obtaining a new configuration described in step 2. The same happens e.g. after the first read action and then the new current state \((5, 3)\) will be further explored. In case a path leading to an illegal state is detected, the failure prediction approach will raise a warning declaring the possible dangerous path (i.e. the consecutive execution of two open actions by component \( P \)). This information will be used by failure avoidance techniques to avoid the potential failure to happen. For instance, the invocation could be blocked by the middleware to avoid that component \( Q \) reaches an unstable state. The explored model will be regenerated in case a component is added or removed.

The failure prediction approach we sketched seems to be well suited for situation in which dynamic and sporadic interactions among components happens. In such a situation, the composition cannot rely on deep static verification techniques and, at the same time, the behaviour that they will execute is often a small subset of the whole potential behaviour. So it seems to be more interesting to permit the integration and to possibly avoid emerging dangerous behaviour. Moreover, the impossibility to easily predict the characteristics of the run-time environment and thus the possibility to evolve the system unpredictably is well managed by the novel composition operator and the use we make of it in our algorithm.

\[
P \circled{1} \xrightarrow{\text{open}} \circled{2} \xrightarrow{\text{read}} \circled{3} \xrightarrow{\text{close}} \circled{4} \xrightarrow{\text{write}} \circled{5} \xrightarrow{\text{close}}
\]

Figure 1: Composition and model exploration for two simple components \( P \) and \( Q \).

\[
P \circled{1} \xrightarrow{\text{open}} \circled{2} \xrightarrow{\text{read}} \circled{3} \xrightarrow{\text{close}} \circled{4} \xrightarrow{\text{write}} \circled{5} \xrightarrow{\text{close}}
\]

Look-ahead = 2

The execution of one exploration step by the run-time checking algorithm is faster than the execution of the corresponding step by the real system. This assumption is necessary to keep the exploration synchronized with the real execution. The release of this assumption would require the introduction of specific mechanisms to keep the monitor synchronized with the running system.

The online failure prediction algorithm we have implemented (described in Section III-C) relies on a specification of the component behaviour based on the interface automata formalism. The algorithm suitably composes the specifications of those components under execution. Instead of using the classical composition operator defined in [6], we base our algorithm on a slightly different composition operator (defined in Section III-B, Def. 3) that, according to us, is more suitable for the online prediction of failures in a dynamic environment. For our composition operator, any couple of components sharing a set of I/O actions can be integrated. Then, the composed automata is navigated by looking ahead to the current execution state. It is the task of our failure prediction approach to check whether the system is approaching an illegal state, and so to inform a possible failure avoidance mechanism that will take care of repair actions. An illegal state corresponds to an integration failure, by any path shorter than \( k \) steps and originating in the current state. In a sense, we assume that an illegal state can be reached as consequence of a wrong invocation/message done by one component on a component that either is not willing to accept it in the current state or does not exist at all.

Figure 1 sketches the general idea of our algorithm in case of two simple components \( P \) and \( Q \). Component \( Q \) models a file in a file system; after its opening, it can be repeatedly read and written until its closure. The automaton \( P \) represents a process that wants to use the resource made available by \( Q \).

First, the specifications of the two executing components are composed. For example, when both components \( P \) and \( Q \) are in state 2, \( P \) can perform an open action; this is a shared output whose corresponding input at the moment is not available. As described above, this invocation causes the evolution of the composed system into the illegal state \((3, 2)\) (see the lower part of Figure 1).

Then, the failure prediction approach we present in this paper explores the composed model as illustrated in the lower part of Figure 1. Starting from the initial state \((1, 1)\), and assuming a look-ahead \( k = 2 \), the failure prediction approach explores the possible future states reachable by any path longer at most 2. The initial configuration (associated to state \((1,1)\)) is described in step 1 of Figure 1. After the open, the failure prediction mechanism will move the explored model of a further step assuming as current state the pair \((2, 2)\) and obtaining a new configuration described in step 2. The same happens e.g. after the first read action and then the new current state \((5, 3)\) will be further explored. In case a path leading to an illegal state is detected, the failure prediction approach will raise a warning declaring the possible dangerous path (i.e. the consecutive execution of two open actions by component \( P \)). This information will be used by failure avoidance techniques to avoid the potential failure to happen. For instance, the invocation could be blocked by the middleware to avoid that component \( Q \) reaches an unstable state. The explored model will be regenerated in case a component is added or removed.

The failure prediction approach we sketched seems to be well suited for situation in which dynamic and sporadic interactions among components happens. In such a situation, the composition cannot rely on deep static verification techniques and, at the same time, the behaviour that they will execute is often a small subset of the whole potential behaviour. So it seems to be more interesting to permit the integration and to possibly avoid emerging dangerous behaviour. Moreover, the impossibility to easily predict the characteristics of the run-time environment and thus the possibility to evolve the system unpredictably is well managed by the novel composition operator and the use we make of it in our algorithm.

B. A Model for Dynamic Composition and Compatibility

In this section we formally define the new product operator on which our algorithm is based on. We then show that the product of mutually composable interface automata is associative (see Prop. 1). This allows us to extend the binary product operator of Def. 3 in order to get the product of \( n \geq 2 \) mutually composable interface automata (see Def. 4).

Definition 3: Let \( P \) and \( Q \) be two composable interface automata and \( A \) a set of actions such that \( A \cap \text{shared}(P, Q) = \emptyset \) (this means that actions in \( A \) can be
shared with $P$ or $Q$, but not with both) and $A \cap A_H^P = \emptyset = A \cap A_H^Q$. The product $P \otimes_A Q$ is the interface automaton we provide in Table I.

We define Illegal($P, Q, A$) = \{ ($v, u$) $\in V_P \times V_Q$ $|$ $\exists a \in A_P^O \cap \langle A_P^I, A_Q^I \rangle (u) \wedge (v, a, v') \in T_P$ $\cup$ \{ ($v, u$) $\in V_P \times V_Q$ $|$ $\exists a \in A_Q^O \cap \langle A_P^I, A_Q^I \rangle (u) \wedge (u, a, u') \in T_Q$ $\cup$ \{ ($v, u$) $\in V_P \times \emptyset$ $|$ $\exists a \in A_P^O \cap \langle A_P^I, A_Q^I \rangle (u) \wedge (v, a, v') \in T_P$ $\wedge$ $\exists a \in A_Q^O \cap \langle A_P^I, A_Q^I \rangle (u) \wedge (u, a, u') \in T_Q$ \}
\}

Intuitively, the set $A$ in $P \otimes_A Q$ is a set of actions that either $P$ or $Q$ shared with a component that has been removed at a previous stage of the system evolution. These are illegal because represent an attempt to communicate with a component that does not exist anymore. A shared output is illegal in $(v, u)$ when it represents an invocation/message on a component that is not willing to accept it.

The proof of the following proposition is available in Appendix A.

**Proposition 1:** Let $P_1, P_2, P_3$ be three interface automata where $P_i = (V_i, V_i^{init}, A_i^I, A_i^O, A_i^H, T_i)$ for each $i$. Let $A$ be a set of actions such that $A \cap \text{shared}(P_i, P_j) = \emptyset = A \cap A_H^P$ for each $i, j$. If $P_1, P_2$, and $P_3$ are mutually composable, then $(P_1 \otimes_A P_2) \otimes_A P_3 = P_1 \otimes_A (P_2 \otimes_A P_3).

**Definition 4:** Let $\{P_i\}_{i=[1,n]}$ be a set of $n \geq 2$ mutually composable interface automata and $A$ a set of actions such that $A \cap \text{shared}(P_i, P_j) = \emptyset = A \cap A_H^P$ for each $i, j$. We define $P = \bigotimes_A \{P_i\}_{i=[1,n]}$ to be the interface automaton we provide in Table II, where $S = \bigcup_{i<j \in [1,n]} \text{shared}(P_i, P_j)$ and, for each $i \in [1,n], A_i = (A_i^I \setminus S) \cup (A_i^O(v_i) \cap \bigcup_{j \in [1,n]} A_j^I \setminus A_j^I(v_j)).$

We can generalize the notion of illegal state as follows: a state $v = (v_1, \ldots, v_n)$ of $P$ is illegal if there exist $i \in [1,n], a \in (A_i^O \cap \bigcup_{j \in [1,n]} A_j^I(v_j)) \cup A$ and $v_i' \in V_i$ such that $(v_i', a, v_i') \in T_i$. Since each action can be shared by at most a pair of interface automata (due to mutual composability), for each $i$ there is at most a $j$ such that $A_i^O \cap (A_j^I \setminus A_j^I(v_j)) \neq \emptyset$.

**C. The Online Failure Prediction Algorithm**

We assume that, at any time, an arbitrary number (let say $n$) of services are running on our platform. Each service is provided with an interface represented by an interface automaton $P_i = (V_i, V_i^{init}, A_i^I, A_i^O, A_i^H, T_i)$, for $i \in [1, n]$.

We also assume that these automata are deterministic and mutually composable.

To describe how $P = \bigotimes_A \{P_i\}_{i \in [1,n]}$ changes over time, we use an arc-labeled oriented graph $T = (V_T, E_T)$ (we call it the exploration graph) whose vertices and edges are nodes and steps of $P$, i.e. $V_T \subseteq V_P$ and $E_T \subseteq V_P \times A \times V_P \subseteq T_P$, resp. The current system configuration is described by means of a special vertex $r = (r_1, \ldots, r_n) \in V_T$. The underlying assumption is that the subgraph of $T$ whose vertices are those reachable from $r$ by a depth-first visit of length $k$ describes how $P$ will evolve in the next $k$ steps; as expected, $k$ is the number of look-ahead steps. At first, $r = (v_1^{init}, \ldots, v_n^{init})$; during the system evolution, each $r_i$ denotes the current state of corresponding $P_i$.

The graph $T$ is initially built by the procedure INIT (Alg. 1). It is then updated whenever an action occurs (see Alg. 4) or the system configuration changes because components are added or removed (see Alg. 5 and 6). INIT initializes $T$ by adding all vertexes (and the corresponding edges) that can be reached from $r = (v_1^{init}, \ldots, v_n^{init})$ in $k$ steps. To this aim, we use an auxiliary procedure ONESTEPFROM($X$) that takes in input a set $X \subseteq V_P$ of vertexes and updates $T$ by considering all possible steps that are enabled at legal states of $X$ (see Alg. 2). Note that, at first $X := \{r\}$; then it contains all vertexes that we can reach from $r$ with $1, 2, \ldots, k$ steps. We omit the description of the algorithm LEGAL($v$) where $v = (v_1, \ldots, v_n)$ $\in V_P$. This algorithm checks if the state $v$ is legal or not according to Def. 4. As expected, it returns false if there are $i, j \in [1, n], a \in (A_i^O \cap (A_j^I \setminus A_j^I(v_j))) \cup A$ and $v_i' \in V_i$ such that $(v_i', a, v_i') \in T_i$. Otherwise, it returns true.

1 **Input:** The set $\{P_i \mid i \in [1,n]\}$ and the number $k \neq 1$ of look-ahead steps
2 **Output:** The initial graph $T = (V_T, E_T)$
3 // both $r$ and $A$ are initialized by Alg. 3
4 $V_T, E_T := \emptyset;
5 X := \{r\};
6 i := 1;
7 while $i < k$ and $X \neq \emptyset$
8 $X := \text{ONESTEPFROM}(X);
9 i := i + 1;
10 end

**Algorithm 1:** INIT()
\begin{eqnarray*}
V_P &=& V_1 \times V_2 \times \cdots \times V_n \\
A_P^I &=& (\bigcup_{i \in [1,n]} A_i^I) \setminus \mathcal{S} \\
A_P^H &=& (\bigcup_{i \in [1,n]} A_i^H) \setminus \mathcal{S} \\
T_P &=& \bigcup_{i \in [1,n]} \{ (v_1, \ldots, v_i, \ldots, v_n), a, (v_1, \ldots, v'_i, \ldots, v_n) \mid (v_i, a, v'_i) \in T_i \land a \in A_i \} \\
& & \cup \\
& & \bigcup_{i, j \in [1,n]} \{ ((v_1, \ldots, v_i, \ldots, v_j, \ldots, v_n), a, (v_1, \ldots, v'_i, \ldots, v'_j, \ldots, v_n)) \mid (v_i, a, v'_i) \in T_i \land (v_j, a, v'_j) \in T_j \land a \in \text{shared}(P_i, P_j) \}
\end{eqnarray*}

Table II: The interface automaton $P = \bigotimes_A \{P_i\}_{i \in [1,n]}$.

\begin{algorithm}
\begin{algorithmic}[1]
1 \textbf{Input:} A set of vertexes $X \subseteq V_T$
2 \textbf{Output:} Update $T$ by adding all steps that are enabled at legal states of $X$.
3 $Y := \emptyset$
4 \textbf{foreach} $v = (v_1, \ldots, v_n) \in X$ \textbf{do}
5 \hspace{1em} \textbf{if} \text{LEGAL}(v) \textbf{then}
6 \hspace{2em} \textbf{for} $i := 1$ \textbf{to} $n$ \textbf{do}
7 \hspace{3em} \textbf{foreach} $a \in A_i$ \textbf{and} $(v_i, a, u_j) \in T_i$ \textbf{do}
8 \hspace{4em} $u := (v_1, \ldots, u_i, \ldots, v_n)$;
9 \hspace{4em} $Y := Y \cup \{u\}$;
10 \hspace{4em} $V_T := V_T \cup \{v\}$;
11 \hspace{4em} $E_T := E_T \cup \{(v, a, u)\}$
12 \hspace{2em} \textbf{end}
13 \hspace{1em} \textbf{end}
14 \hspace{1em} \textbf{if} \exists i, j \in [1,n] \text{ s.t. } a \in \text{shared}(P_i, P_j) \textbf{then}
15 \hspace{2em} \textbf{foreach} $(v_i, a, u_j) \in T_i$ \textbf{and} $(v_j, a, u_j) \in T_j$ \textbf{do}
16 \hspace{3em} $v := (v_1, \ldots, u_i, \ldots, u_j, \ldots, v_n)$;
17 \hspace{3em} $Y := Y \cup \{v\}$;
18 \hspace{3em} $V_T := V_T \cup \{v\}$;
19 \hspace{3em} $E_T := E_T \cup \{(v, a, u)\}$
20 \hspace{2em} \textbf{end}
21 \hspace{1em} \textbf{else}
22 \hspace{2em} \text{\textsc{WARNING}("Illegal state:" + v)}
23 \hspace{2em} \textbf{end}
24 \hspace{1em} \textbf{end}
25 \textbf{return} $Y$
\end{algorithmic}
\caption{\textsc{OneStepFrom}(X)}
\end{algorithm}

Our algorithm for failure prediction (Alg. 3), repeatedly waits for an event $e$. If $e$ corresponds to the insertion and removal of a component, the exploration graph $T$ is consistently updated (see Alg. 5 and 6). In case of an insertion, we assume that the interface automaton associated with a new component is mutually composable with the automata we are already monitoring. The set $A$ of actions that are illegal due to the changes of context is initially set to be empty in Alg. 3 and later modified by Alg. 5 and 6. The insertion of a new component $Q$ removes from $A$ all actions of $Q$ itself, while the elimination of the $i$-th components adds to the set $A$ all the actions that $P_i$ has shared with some other component (at most one, due to mutual compositability). To be sure that the product is still associative, we have also proved that the property $A \cap \text{shared}(P_i, P_j) = \emptyset = A \cap A_i^H$ for each $i, j$ is preserved by these changes. Finally note that both $\text{ADD}(Q)$ and $\text{REMOVE}(i)$ reconstruct the graph $T$ from the beginning.

If $e$ corresponds to a monitored action $a ^ 3$, we consider the vertex $v$ reached from $r$ by means of $a$ and then explore the graph $T$ to check whether (and, in this case, how) an illegal state is reached along the next $k$ execution steps from $v$, see Alg. 4. At this stage, we also extend $T$ further by adding all states that we can enter from the set of vertexes that are reachable from $v$ in exactly $k-1$ steps (to get this set we use an auxiliary procedure $\text{REACHABLEFROM}$; it is essentially a depth-first visit starting at $v$ and, hence, quite expected).

Note that, whenever an illegal state is encountered, we simply raise a warning. A real implementation could use this information e.g. to start a recovery procedure and try to solve the problem. Otherwise, if no illegal states are encountered, we can consider this $a$ ‘safe’ and freely perform it. Clearly, to improve our level of trust in performing this $a$, we must increase the number of look-ahead steps $k$.

\begin{algorithm}
\begin{algorithmic}[1]
1 \textbf{Input:} A set $\{P_i \mid i \in [1,n]\}$ of $n$ mutually composable interface automaton, the number $k \in \mathbb{N}$ of look-ahead steps
2 $r := (v_1^{\text{init}}, \ldots, v_n^{\text{init}})$;
3 $A := \emptyset$;
4 $\text{INIT}(r)$;
5 \textbf{while} true \textbf{do}
6 \hspace{1em} wait until an event $e$ occurs
7 \hspace{2em} \textbf{switch} $e$ \textbf{do}
8 \hspace{3em} \textbf{case} $e = a \in \bigcup_{i \in [1,n]} A_i$
9 \hspace{4em} $\text{EXPLORE}(a)$;
10 \hspace{3em} \textbf{case} $e = \text{ADD}(Q)$
11 \hspace{4em} $\text{ADD}(Q)$;
12 \hspace{3em} \textbf{case} $e = \text{REMOVE}(i)$
13 \hspace{4em} $\text{REMOVE}(i)$
14 \hspace{2em} \textbf{end}
15 \hspace{1em} \textbf{end}
\end{algorithmic}
\caption{An algorithm for online failure prediction}
\end{algorithm}

3 Recall that we assume that OSGi can monitor every action of each automaton $P_i$, also the internal ones.
AUTOMATA (in XML format), and some aspectual codes.

OSGi platform following the main scenario described below.

proven implementation published by the Eclipse Foundation,
specification, the one we are using is Equinox [10], a well-
designed to the platform without affecting the running services.

for deployment) can be remotely installed, started, stopped,
updated and uninstalled without requiring a reboot. For this
purpose, each bundle is loaded with a different class loader
added or removed, the bundles’ interface automata are
passed to this CASSANDRA-specific bundle so to monitor
the global system behavior. Thanks to this, the execution of
any action defined within an interface automata is notified
to the CASSANDRA's bundle to permit the exploration of
the composed model. While it would be more efficient to
embed the monitoring and prediction strategy directly within
the OSGi infrastructure, we decided to extend the current
infrastructure with wrappers and specific bundles so to avoid
to create an ad hoc OSGi implementation.

IV. CASSANDRA CONCRETIZATION IN OSGI

In OSGi [9], components (coming in the form of bundles
for deployment) can be remotely installed, started, stopped,
updated and uninstalled without requiring a reboot. For this
purpose, each bundle is loaded with a different class loader
and the “Hot-Deployment feature” enables to add new bund-
dles to the platform without affecting the running services.
While there are a number of implementations of the OSGi
specification, the one we are using is Equinox [10], a well-
proven implementation published by the Eclipse Foundation,
which offers all the necessary features that we require. The
CASSANDRA approach has been implemented within the
OSGi platform following the main scenario described below.

The OSGi bundle developer provides a wrapper for each
bundle, containing the bundle source codes, its interface
automata (in XML format), and some aspectual codes.

A mapping between the source codes and the interface
automata model has to be defined. For this purpose, we
recommend to map a private method of the source code into
an internal action in the interface automata model, while a
public or protected method has to be mapped into an input
or out action in the interface automata model.

The wrapper contains also aspect-based source codes.
They are necessary to instrument the bundle at run-time with
mechanisms for notifying the invocations the bundle will
receive, and to expose the interface automata to be used by
an external analyzer. This is possible thanks to the Equinox
Aspects [11] technology that integrates aspect weaving,
using AspectJ, into the Equinox OSGi implementation.

The online failure prediction algorithm defined in Sec-
tion III is included within a specific OSGi bundle that
provides a system monitoring and diagnostic service. When-
ever the system is started up, or when a component is
added or removed, the bundles’ interface automata are
passed to this CASSANDRA-specific bundle so to monitor
the global system behavior. Thanks to this, the execution of
any action defined within an interface automata is notified
to the CASSANDRA's bundle to permit the exploration of
the composed model. While it would be more efficient to
embed the monitoring and prediction strategy directly within
the OSGi infrastructure, we decided to extend the current
infrastructure with wrappers and specific bundles so to avoid
to create an ad hoc OSGi implementation.

V. CASE STUDY AND EVALUATION

This section, after introducing the Toast OSGi application,
will evaluate CASSANDRA and briefly discuss advantages
and open issues.

A. The Toast OSGi Application

Toast [7] is an example application meant to demonstrate
a wide range of EclipseRT technologies. It provides a system
interface to devices installable in a vehicle, and a user
interface for interacting with or managing them.

At a high level, Toast consists of a client and a back
end. In the simplest initial configuration the Toast Client
only provides an emergency application that notifies the
back end of the vehicle’s GPS location when the airbag
deploys. The Toast back end is developed from a simple
emergency station. Starting from the basic configuration,
new components and features can be installed at run-time
during system operation, like, Google Earth, weather fore-
cast, an application that tracks the vehicle’s GPS location,
a touch screen interface to control the vehicle’s audio and
climate systems, a turn-by-turn navigation system, and a
more complex back end that communicates with an OSGi-
based fleet management control center using a variety of
protocols.

Toast is fully implemented as an OSGi application. Its full
source codes can be downloaded here [12]. Eleven different

1. **Input**: An action $a \in \bigcup_{i \in [1,n]} A_i$
2. Let $v \in V_T$ such that $(r, a, v) \in E_T$;
3. if $\text{LEGAL}(v)$ then
   1. $X := \text{REACHABLEFROM}(v, k - 1)$;
   2. if $X \neq \emptyset$ then
      1. $r := v$;
      2. $X := \text{ONESTEPFROM}(X)$;
   3. else Warning("No states can be reached in k steps");
4. end
5. else Warning("Illegal state: " + v)
6. end
7. Algorithm 4: EXPLORE($a$)

1. **Input**: An interface automaton $Q$
2. $n := n + 1$;
3. $P_n := Q$;
4. $r := (r_1, \ldots, r_n, v_0^n)$;
5. $A := A \setminus A_Q$;
6. INIT()
7. Algorithm 5: ADD($Q$)

1. **Input**: An index $i \in [1,n]$ that corresponds to the
   component we are removing
2. for $j := i$ to $n - 1$ do
   1. $P_j := P_{j+1}$
3. end
4. $n := n - 1$;
5. $r := (r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_n)$;
6. $A := A \cup \bigcup_{j \in [1,n]} \text{shared}(P_i, P_j)$;
7. INIT()
8. Algorithm 6: REMOVE($i$)
services can be added to the basic configuration. The related bundles (summarized in Table III) can be added into the system in any different order. We created a wrapper for each bundle, each including the interface automaton and the aspect codes associated to each bundle.

The four bundles in the Toast initial configuration can be modelled by the four interface automata in Figure 2. When the airbag is deployed (i.e. $P_3$ performs the internal action `deploy`), the bundle AIRBAG sends a message (via the action `em`) to the emergency application and waits for an acknowledge (an `emAck` action). In case of a negative replay (modelled by action `emNoAck`), AIRBAG sends the emergency message again. Afterwards, EMERGENCY first communicates with the bundle GPS to get vehicle’s latitude, longitude, head and speed; then it sends a message with all the GPS information to the back end emergency application. Finally, when the airbag is undeployed, AIRBAG sends a corresponding message to EMERGENCY allowing the system to come back to its initial state. Note that EMERGENCY can also only interact with the GPS and the back end emergency.

Figures 3 and 4 describe some steps of our algorithm. At first, all components are available; so, the initial graph $T_0$ and the graph $T_1$ after a `deploy` action are given in Figure 3. Each vertex in these graphs is a tuple in $V_1 \times V_2 \times V_3 \times V_4$; moreover $A = \emptyset$. This means that illegal states can only be reached via illegal shared outputs, as e.g. $em$ in the states $(5, 0, 1, 0)$ and $(6, 1, 1, 0)$. In both these states bundle EMERGENCY is not able to accept $em$ because `runEmProc` has already been performed.

Figure 4 describes how $T_1$ changes if, whenever in $(0, 0, 1, 0)$, the bundle GPS is removed (assume e.g. that this

### Table III: The #states and #transitions of different bundles.

<table>
<thead>
<tr>
<th>Configurations</th>
<th>Bundles</th>
<th>#States</th>
<th>#Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Initial</td>
<td>clientemergency</td>
<td>29</td>
<td>38</td>
</tr>
<tr>
<td></td>
<td>dev.gps</td>
<td>15</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>dev.airbag</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>backend.emergency</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2. Backend Portal</td>
<td>backend.portal</td>
<td>14</td>
<td>25</td>
</tr>
<tr>
<td>3. Tracking</td>
<td>client.tracking.config</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>client.tracking</td>
<td>12</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>backend.tracking</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>client.nav.guidance</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>client.nav.mapElement</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>client.nav.routeData</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>client.nav.routeSegment</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>client.nav.segmentData</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>client.nav.segmentContext</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5. Control Center</td>
<td>backend.controlLocation</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>backend.controlcenter</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>backend.vehicle</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>6. Amplifier</td>
<td>dev.amplifier</td>
<td>39</td>
<td>40</td>
</tr>
<tr>
<td>7. CdpLayer</td>
<td>dev.cdpLayer</td>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>8. Climate</td>
<td>dev.climate</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>9. Radio</td>
<td>dev.radio</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>10. Audio</td>
<td>dev.audio.radio</td>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>dev.audio.audioScreen</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>dev.audio.radioController</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>dev.audio.radioSubscreen</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>11. Google Map</td>
<td>dev.google</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>12. Crust</td>
<td>crust.display</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>crust.shell</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>crust.widgets</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
component is not available due to a system fault). Vertices in this graph are tuples in $V_1 \times V_2 \times V_3$; moreover now $A = \{ X, getX, XErr \}$ where $X \in \{ Lat, Long, Head, Speed \}$. In contrast to Figure 3, the execution of these actions causes the system to be in an illegal state, as e.g. $(6,1,0)$ and $(6,5,0)$. Its worth noting that, when GPS will be added back to the system, it will become again possible to perform actions in $\{ X, getX, XErr \}$ without moving in an illegal state because Alg. 5 will remove them from $A$.

### B. Experimental validation

This section will carry out a series of experiments to measure the performance of our theory and algorithm. The basic research questions are designed as follows:

- **RQ1**: How long it takes for the online failure prediction algorithm to construct the future $k$-steps graph?
- **RQ2**: How much does the online failure prediction algorithm affects the system overhead?

**Experiment setup**: In order to answer these questions, we smoothly extended our previous tool [13] into the OSGi-based platform and fully implemented the Toast case study. As a platform for our experiments we used a Windows-based PC equipped with a 1.83 GHz dual-core processor and 2GB of main memory.

The **RQ1** can be divided into three sub-questions:

- **RQ1.1**: How long it takes for the proactive monitoring algorithm to construct the initial graph with $k$ steps ahead?
- **RQ1.2**: How long it takes for the algorithm to construct the graph during system execution (e.g., for execution states different from the initial one)?
- **RQ1.3**: How long it takes for the algorithm to be applied when run-time system evolution happens?

For **RQ1.1**, we carried out two types of experiments. Firstly, we recorded the execution time for loading the exploration graph from the initial state because Alg. 5 will remove them from $A$. For example, the time increased to 399548.9ms when the system is in $conf12$ and the look ahead step is set to 50. Consequently, when the look ahead step is big enough and the composition model is also large, the algorithm may suffer from the state space explosion problem. It is indeed true that, while for experimental purposes using big $k$ makes sense, we do not expect online failure prediction techniques to be run with big values (as advocated by [3], online predictions are reliable only when the considered time frame is limited). Moreover, while the construction of the initial graph may require some time, this calculation needs to be done only once, when the system is in its initial configuration.

For question **RQ1.2**, we again record the execution time for re-constructing the exploration graph when the system evolves, and with different $ks$. Chart3 in Figure 5 shows the results. The time increasing is quite limited at this time, with a pick of 370ms at step 50. This is due to the fact that there is no need to reconstruct the whole model: when the new initial state is detected, only one further exploration step is required. From the experiments, we can conclude that the re-construction of the graph (after its initial creation) will happen in a limited time. This result supports our claim that **CASSANDRA** can be potentially applied to real systems.

For question **RQ1.3**, we record the time for re-constructing the graph model when the system evolves. When we set $k = 15$, the time for re-constructing the run-time model is about 130ms in the initial configuration. When the system evolves from $conf1$ to $conf2$, the algorithm needs to take more time to update and re-construct the whole run-time model. Consequently, the time will increase to more than 400ms. Then, the run-time re-constructing algorithm is used again. So the time decrease to around 200ms. Then the time will again increase to around 450ms and decrease to 240ms when the system evolves from $conf2$ to $conf3$. From the experiments we can conclude that the time will
first increase and then decrease to some degree when run-time evolution happens. The time increases due to the fact that it will take much time to update and re-construct the whole run-time model. The time then decreases due to the partial re-constructing run-time models.

To answer RQ2, we run both the original system and the one with the monitoring features so to measure the overhead. JConsole [14] is used for this purpose. We compare two cases when $k = 10$ and $k = 30$. Chart5 in Figure 5 shows the overhead for memory used. Our experiments show that the original system uses 3.6Mb while the monitored one has a memory overhead of around 19% at the beginning. When the system evolves from config1 to config2, the memory overhead increases to 23%, due to the graph re-construction. Then, the overhead decreases to some degree. From the figure we can see that similar conditions happen when the system evolves from config2 to config3, and from config3 to config4. Then, we set $k = 30$ in the next experiment. Initially, the overhead for memory usage is around 22%. It increases to 35%, then decreases to 27% when the system evolves from config1 to config2. Since $k$ is increased, the overhead increases more than before. When the system evolves from config3 to config4, the memory usage increases dramatically to 48%. Chart6 in Figure 5 shows the CPU overhead. Similar conditions happen. When $k = 10$ the CPU overhead decreases a little while it will increase dramatically to around 50% when $k = 30$.

C. (Some) Considerations

Apart from the assumptions described in Section III-A, the approach has the following limitations:

- **Interface Automata consistency with OSGi bundles.** In the Toast study, interface automata are created so to reflect the source code behavior. As always happens with specifications, manually building interface automata specifications for complex OSGi bundles may become tedious and error-prone. Existing static specification mining approaches (e.g.,
(15)) may be potentially adapted to automatically generate the automata models from an OSGi bundle.

- **Applicability to other component models.** Interface automata cannot be seen a standard language to specify components behavior. In a component model like Fractal [16] that uses a recursively and reflective model, for example, interface automata is not an appropriate specification language.

- **Right steps for looking ahead when evolution happens**
  When evolution happens, we may want to be able to adapt the look ahead value $k$ as well, so that when the graph becomes larger we reduce $k$ to manage complexity, while when the graph reduces in complexity, we increase $k$ so to predict earlier future failures.

- **Component specification.** We make the assumption that a specification of each component is always available, and this is certainly not always true.

VI. RELATED WORK

This section provides related work on failure prediction, and run-time verification of dynamically evolving systems.

**Failure Prediction.** The most significant paper on online failure prediction can be most probably considered the survey conducted by Salfner et al. in [3]. This survey analyzes and compares a number of existing online failure prediction methods. First, a taxonomy is proposed in order to structure and classify the existing online failure prediction methods. Then, forty seven online prediction methods are considered and mapped into the taxonomy. Most of them use heuristics, statistics or probabilistic models to predict failures that may potentially happen in the near future. Metrics are also defined to be able to compare the failure prediction accuracy of the surveyed approaches. Although CASSANDRA belongs to the online failure prediction research branch, differently from the surveyed approaches we specialize on component-based, dynamically evolving systems.

In [17] the authors discuss how to use model checking techniques for discovering defects before they happen. While in general related to our approach, this work does not focus on failure prediction of dynamically evolving systems.

Vieira et al. [18] discuss on how to improve and validate failure prediction methods using fault injection. The main focus of this paper is on demonstrating how the injection of software faults is a powerful tool to validate and improve failure prediction mechanisms. While not directly related to what presented in this paper, fault injection techniques may be studied in future work in order to further evaluating CASSANDRA.

Epifani et al. [19] propose an approach where run-time models are analyzed to detect or predict if a desired property is, or will be, violated by the running implementation. Similarly to CASSANDRA, this approach deals with dynamically evolving systems. Differently, it focuses on non-functional properties.

In a previous paper by the authors [20], a sketch of the CASSANDRA algorithms and its initial customization to OSGi has been presented. Differently, this work provides a detailed and formal description of the algorithms, revises them in order to make them computationally faster, applies the theory to a case study, and empirically evaluates CASSANDRA.

**Run-time Verification of Dynamically evolving systems.** A considerable number of approaches have been developed for run-time monitoring of dynamically evolving systems (e.g., [21], [22], [23], [24], [25]).

The authors in [21] show how it is possible to generate a snapshot of the structure of a running application, and how this can be combined with behavioral specifications for components to check compatibility against system properties. Barringer et al. [22] describe mechanisms for combining programs from separate components and an operational semantics for programmed evolvable systems. Goldsby et al. [23] propose a run-time monitoring and verification technique that can check whether dynamically adaptive software satisfies its requirements. Baresi et al. [24] propose Dynamo, a simple architecture that, through specific and simple annotations, enables the automatic creation of instrumented WS-BPEL processes that can be monitored. The level of monitoring can be dynamically set through a web-based interface. Muccini et al. [25] propose the MOSAICO approach to monitor (kernel-based) evolving component-based systems and verify interaction pattern properties.

While those approaches verify dynamically evolving systems, none of them implement failure prediction features.

VII. CONCLUSIONS AND FUTURE WORK

This paper has proposed CASSANDRA, a novel proactive monitoring approach for online failure prediction of dynamically evolving systems. The approach captures the current state of the system through proactive monitoring its execution, and uses it to explore design-time system models built on-the-fly. The approach has been initially formalized through different algorithms, and successively customized into the OSGi component framework. Some evaluations have been performed to show the feasibility of the proposed algorithms and tools on the Toast case study.

A substantial list of future works for CASSANDRA is in our wish list: i) we are working on an extension of the approach to consider general temporal properties. Obviously given its characteristics CASSANDRA will only be able to detect violation of safety properties and bounded liveness properties where the look-ahead value is greater than the bounding value; ii) compare our solution with state-of-the-art model checking tools to show the significant benefits; iii) study how much to look ahead (i.e., how to set-up the look ahead parameter properly) according to different types of
systems, in order to derive a completely useful theory that could be used to guide how to monitor real systems.

REFERENCES


APPENDIX A: PROOF OF PROPOSITION 1

Proposition 1 Let $P_1, P_2, P_3$ be three interface automata where $P_i = (V_i, V^\text{init}_i, A_i^\text{O}, A_i^\text{H}, T_i)$ for each $i$. Let $A$ be a set of actions such that $A \cap \text{shared}(P_1, P_2) = \emptyset = A \cap \text{shared}(P_1, P_3)$ for each $i$. If $P_1, P_2, P_3$ are mutually composable, then $(P_1 \otimes A P_2) \otimes A P_3 = P_1 \otimes A P_2 \otimes A P_3$.

Proof: We write $P_12$ and $P_23$ to denote, resp., $P_1 \otimes A P_2$ and $P_2 \otimes A P_3$; moreover, $P = P_12 \otimes A P_3$ and $Q = P_1 \otimes A P_2$. If $P_1, P_2$ and $P_3$ are mutually composable and, hence, $A_i^\text{H} \cap A_j = \emptyset$ and $A_i^\text{H} \cap A_j = \emptyset = A_i^\text{O} \cap A_j^\text{O}$ for each $i \neq j$, then:

- $\text{shared}(P_1, P_2)$, $\text{shared}(P_1, P_3)$ and $\text{shared}(P_2, P_3)$ are mutually disjoint sets of actions (i.e. each action is shared by at most two automata). Assume, toward a contradiction, that there is an $a \in \text{shared}(P_1, P_2) = (A_i^\text{H} \cap A_j^\text{O}) \cup (A_i^\text{O} \cap A_j^\text{H})$ such that $a \in \text{shared}(P_1, P_3) = (A_i^\text{H} \cap A_j^\text{O}) \cup (A_i^\text{O} \cap A_j^\text{H})$. Then either $a \in A_i^\text{H} \cap A_j^\text{O}$ or $a \in (A_i^\text{O} \cap A_j^\text{H})$. From now on, we write $S$ do denote $\text{shared}(P_1, P_2) \cup \text{shared}(P_1, P_3) \cup \text{shared}(P_2, P_3)$.

- $A_i^\text{H} \cap \text{shared}(P_j, P_k) \subseteq A_i^\text{H} \cup (A_j^\text{O} \cup A_k^\text{O}) = \emptyset$. Similarly, it is $A_i^\text{O} \cap \text{shared}(P_j, P_k) = \emptyset$, and, since $A_i^\text{H} \cap \text{shared}(P_j, P_k) = \emptyset$, $A_i^\text{H} \cap \text{shared}(P_j, P_k) = \emptyset$. 


This will also imply that \( P_2 \) and \( P_3 \) (as well as \( P_1 \) and \( P_23 \)) are also composable.

- \( A'_2 \cap A'_3 = (A'_1 \cup A'_2) \setminus \text{shared}(P_1, P_2) \cap A'_3 = (A'_1 \cup A'_2) \cap A'_3 \) since \( a \in A'_2 \) implies \( a \notin \text{shared}(P_1, P_2) \). Similarly, \( A'_{12} \cap A'_3 = (A'_1 \cup A'_2) \cap A'_3 \) and, hence, \( \text{shared}(P_{12}, P_3) = ((A'_1 \cup A'_2) \cap A'_3) \cap A'_3 = \text{shared}(P_{12}, P_3) \cup \text{shared}(P_2, P_3) \). In the same way we can prove that \( \text{shared}(P_{12}, P_3) \).

- \( A'_2 \cap A'_3 = (A'_1 \cup A'_2) \setminus \text{shared}(P_{12}, P_3) \) since \( a \in A'_3 \) implies \( a \notin \text{shared}(P_{12}, P_3) \). Thus, \( A'_2 = (A'_1 \cup A'_2) \setminus \text{shared}(P_{12}, P_3) \). Similarly, we can prove \( A'_1 = (A'_1 \cup A'_2) \setminus \text{shared}(P_{12}, P_3) \) and \( A'_3 = (A'_1 \cup A'_2) \setminus \text{shared}(P_{12}, P_3) \).

- \( A''_2 = A''_1 \cup A''_2 \cup \text{shared}(P_{12}, P_3) = (A''_1 \cup \text{shared}(P_{12}, P_3)) \cup \text{shared}(P_{12}, P_3) = A''_1 \cup \text{shared}(P_{12}, P_3) \cup \text{shared}(P_{12}, P_3) \). Likewise it is \( A''_3 = A''_1 \cup A''_2 \cup \text{shared}(P_{12}, P_3) \).

Thus:

- \( V_P = V_Q = V_1 \times V_2 \times V_3 \)
- \( V'_{\text{init}} = V''_{\text{init}} = V'_3 \times V''_{\text{init}} \times V'_3 \)
- \( A'_2 = A'_1 \cup A'_2 \cup A'_3 \)
- \( A'_3 = A'_1 \cup A'_2 \cup A'_3 \)
- \( A''_2 = A''_1 \cup A''_2 \cup A''_3 \)
- \( A''_3 = A''_1 \cup A''_2 \cup A''_3 \)

It still remains to prove that \( T_P = T_Q \). Let \( v = (v_1, v_2, v_3) \) and \( u = (u_1, u_2, u_3) \) be two states in \( V_P = V_Q \); below, we show that \( (v, a, u) \in T_P \) if \( (v, a, u) \in T_Q \) by considering the following possible cases:

- \( a \in \text{shared}(P_1, P_3) \) and, hence, \( a \in \text{shared}(P_{12}, P_3) \).
- \( a \notin \text{shared}(P_1, P_3) \) and \( a \notin A_2 \) (see above).

There are three possible subcases to examine.

1. Let us first assume \( a \in A_1 \cap A_3 \). In this case, \( (v, a, u) \in T_P \) and \( (v, a, u) \in T_Q \) if \( (v, a, u) \in T_1 \) and \( (v, a, u) \in T_2 \).

2. If \( a \in A_1 \cap (A_2 \cap A_3) \), then \( a \notin \text{shared}(P_1, P_2) \) and \( A'_2 = (A'_1 \cup A'_2) \setminus \text{shared}(P_1, P_2) \) imply \( a \in A'_2 \cap (A'_1 \cap A'_3) \). Moreover, \( A'_2 \setminus (A'_2 \cup A'_3) = \text{shared}(P_2, P_3) \). Since \( a \notin \text{shared}(P_2, P_3) \), we have also \( a \in A'_2 \cap (A'_2 \cap A'_3) \).

3. Finally, we assume \( a \in A_3 \cap \text{shared}(A_1, A_3) \).

As above we can prove that it is also \( a \in A_2 \cap (A_1 \cap A_2) \) and \( a \in A_2 \cap (A_1 \cap A_2) \). As a consequence, \( (v, a, u) \in T_P \) and \( (v, a, u) \in T_Q \) if \( u_1 = v_1 \) and \( u_2 = v_2 \) and \( u_3 = v_3 \).

- \( a \in \text{shared}(P_1, P_3) \) and, hence, \( a \in \text{shared}(P_{12}, P_3) \).
- \( a \in A_2 \cap A_3 \). In this case, \( (v, a, u) \in T_P \) and \( (v, a, u) \in T_Q \) if \( u_1 = v_1 \) and \( (v_2, a, u_2) \in T_2 \) and \( (v_3, a, u_3) \in T_3 \).