POLITECNICO DI MILANO
DEPARTMENT ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOCTORAL PROGRAMME IN INFORMATION TECHNOLOGY

DEALING WITH INCOMPLETENESS IN AUTOMATA BASED MODEL CHECKING

Doctoral Dissertation of:
Claudio Menghi

Supervisor:
Prof. Carlo Ghezzi

Co-supervisor:
Prof. Paola Spoletini

Tutor:
Prof. Luciano Baresi

The Chair of the Doctoral Program:
Prof. Carlo Fiorini

2015 – XXVIII

To my grandfather Emilio,
for teaching me not to give up.

To my girlfriend Maria,
life is just a matter of love.

Acknowledge

My PhD adventure started in Limerick (Ireland) while I was sitting in front of a Kebab on a bus directed to the University. I would like to thank the person that, in that moment, talked me for the first time about PhD. I would like to thank all the LERO (the Irish software research centre) group; starting the PhD was a direct consequence of the nice experience I had in Ireland. I would like to thank Maria, for always suggesting me to follow what I like to do. Without these people my Phd would probably not be started.

I would like to express my deepest gratitude to my supervisor, Prof. Carlo Ghezzi, for accepting me as a PhD student. His excellent guidance, caring and patience, provided me a nice atmosphere for doing research. Prof. Ghezzi supported my Phd experience, moulded my mind, and patiently corrected this thesis. A special thank to Prof. Paola Spoletini, my Cicerone. She carried me across the thrilling and engrossing word of research. Unforgettable are our discussions on the semantic of logics over incomplete automata.

How to not remember Politecnico di Milano, all the other PhD students and professors of the Deepse (DEpendable Evolvable Pervasive Software Engineering) group for the dynamic and supportive work environment. An acknowledgement is for “my students", the teaching activity was an amazing experience which offered me the possibility to interact and discuss with their new “fresh minds".

I extend an especially grateful greeting to all the professors that with passion and dedication taught me in the primary and secondary school as well as in the university: everything I learned in my studies helped me in the development of this thesis.

I am indebted to my friends and all people who supported me in different ways during my studies. I want to remember my fellow students and the homeland friends. A special thank to Alessandra, for sharing the morning coffee and her guidance in the contorted world of bureaucracy.

I would like to thank my family, who have always encouraged and believed in me.

Finally, it is with immense gratitude that I acknowledge Maria, the real creator of this thesis. Without her support, great patient and unwavering love the development of this thesis would not be possible.

Abstract

THE software development process concerns a sequence of design decisions that transforms the initial, high-level model of the system into a fully detailed and verified implementation. This process is based on an iterative decomposition of the model of the system into smaller functionalities. At each iteration, the developer may leave the system deliberately incomplete. Incompleteness arises when the design of some functionalities is postponed to a later development stage or when they are left unspecified (e.g., if they will be developed by third parties). The final, fully specified, model of the system is obtained by refining the incomplete parts. The development process concerns the iterative substitution of the postponed functionalities with sub-modules (replacements) and, eventually, executable code. During this process alternative replacements are often explored to evaluate their tradeoffs. It can also happen that replacements of postponed functionalities are only detected at run time, as in the case of adaptive systems.

Formal verification has now become mature. Several techniques [9, 33] allow the designer to check whether the model of the system under development possesses the properties of interest. Formal verification has already been used in practice in several application domains and has also been adopted in many industrial settings. However, there is still a gap between the model checking techniques and the currently used software development lifecycles.

Model checking is a technique that found a prominent role in formal verification. Given a model of the system and a formal property, model checking exhaustively analyzes the model of the system to ensure that all of its behaviors satisfy the property of interest. If the property is not satisfied a counterexample is also returned. Mainstream model checking techniques assume that the model of the system and the properties against which it should be verified are completely defined when the verification takes place. However, this assumption is not always valid during the software development process, as we discussed earlier, since models are often incomplete.

To support *continuous verification* during the development process, we should be able to verify incomplete models. By checking incomplete models even initial, incomplete, and high-level descriptions of the system can be verified against their properties,

supporting an early error detection. Furthermore, when an incomplete part is refined by designing the corresponding replacement, it is desirable to check only the modified part and do not perform the whole verification process from scratch. This approach may distribute the verification effort more uniformly over development and enable also runtime verification, as required by adaptive systems, where certain functionalities only become available while the system is running. In this case, it necessary to check if the functionalities possess certain properties before linking them to the running system.

This thesis aims to provide techniques that support the analysis of incomplete models designed in the software development. First, it proposes Incomplete Büchi Automata (IBAs) a novel modeling formalism that natively supports incompleteness as in the case of top-down development. IBAs extend the well known Büchi automata (BAs) with unspecified states, called black box states, which encapsulate unspecified functionalities. Black box states can be (recursively) refined into other (Incomplete) Büchi automata. In order to analyze IBAs, we propose an *automata-based model checking technique* to verify if an IBA $\mathcal{M}$ satisfies a property ϕ, written in Linear Temporal Logic (LTL). Due to the presence of black box states, the model checking procedure is modified to produce three different values: yes, if the model of the system satisfies its property; no (plus a counterexample), if it does not; unknown, when the property is possibly satisfied, i.e., its satisfaction depends on the replacements, still to be designed, associated to the black box states. Whenever the property is possibly satisfied, a *constraint synthesis* procedure, which is presented in this thesis, allows the computation of a constraint for the unspecified parts. A constraint concerns a set of sub-properties that must eventually be satisfied by the automata fragments (replacements) that will replace the black box states in the refinement process. The developer may use these sub-properties as guidelines in the replacement design. Finally, the thesis presents a *replacement checking* procedure able to verify a replacement against the previously generated constraint. In this way, at each development step, only the new increment is considered in the verification activity.

The approach presented in this thesis has been implemented in the `CHIA` (CHecker for Incompete Automata) framework[1]. `CHIA` is a prototype tool which supports the designer in the system development and its verification and it has been used to evaluate the approach over two practical examples. The first is a classical computer science example and concerns the well known mutual exclusion system. The mutual exclusion system has considered in several works, such as [9, 104]. The second example has been described in [139] and concerns the evolution of a Pick and Place Unit (PPU). The PPU example is used to compare tools that analyze the evolution of automation systems. It is a limited size example, but it provides a valuable trade-off between complexity and evaluation effort [79].

Finally, to analyze the scalability of the approach, in absence of a realistic benchmark suite, the thesis considers a set of random models with increasing size. The evaluation compares the difference in terms of time and space between checking the replacement against the previously generated constraint (the corresponding sub-property) and the effort required to verify the refined model (the original model in which the new component is injected) against the original property.

[1]The tool is available at `https://github.com/claudiomenghi/CHIA/`.

Sommario

Il processo di sviluppo del software è costituito da un insieme di scelte progettuali mediante le quali il modello iniziale del sistema viene iterativamente modificato fino ad ottenere l'implementazione finale. Ad ogni raffinamento lo sviluppatore può lasciare parti del sistema non specificati. In questi casi la specifica viene definita *incompleta*. L'incompletezza sorge quando il raffinamento di alcune funzionalità viene posticipato ad una successiva fase dello sviluppo, o quando i componenti vengono sviluppati da terze parti. Il modello finale del sistema viene ottenuto mediante una sequenza di raffinamenti successivi attraveso i quali le funzionalità identificate vengono iterativamente raffinate in sottomoduli, ed infine rimpiazzate dal corrispettivo codice eseguibile. Durante questo processo varie funzionalità vengono analizzate e confrontate per valutare le loro proprietà. A volte alcune funzionalità possono anche essere aggiunte a run-time, come nel caso dei sistemi adattativi.

Le tecniche di *verifica formale* sviluppate negli ultimi anni [9,33] consentono di verificare se il sistema sviluppato possiede un insieme di proprietà definite dallo sviluppatore. La verifica formale è utilizzata in pratica in vari domini applicativi e viene utilizzata in diversi contesti industriali. Tuttavia, la verifica formale, ed in particolare le tecniche di model checking, non sono completamente integrate nei processi di sviluppo software correntemente utilizzati.

Le tecniche di *model checking*, dato un modello del sistema e una proprietà di interesse, analizzano esaustivamente tale modello per garantire che tutte le sue esecuzioni soddisfino la proprietà. Se la proprietà non è soddisfatta viene generato un controesempio. Nelle tecniche di model checking convenzionali il modello del sistema e le proprietà di interesse sono completamente specificate al momento dell'esecuzione della procedura di verifica. Tuttavia, questa assunzione non è sempre valida, infatti, come discusso in precedenza, i modelli considerati sono spesso incompleti.

Per consentire una *verifica continua* durante il processo di sviluppo del software è necessario fornire delle tecniche capaci di supportare dei modelli incompleti. La verifica di modelli incompleti permette di considerare descrizioni di alto livello del sistema, e di rilevare in anticipo eventuali errori. Inoltre, quando le parti incomplete vengono raffinate e rimpiazzate dai rispettivi componenti, è auspicabile la verifica della sola

parte modificata, in modo da evitare l'esecuzione della procedura di verifica sull'intero modello. Questo approccio consente di distribuire la verifica in maniera più uniforme nel corso del processo di sviluppo software, riducendo significativamente i tempi di verifica e supportando la verifica a run-time. Nella verifica a run-time, richiesta per esempio nel contesto dei sistemi adattativi, è necessario verificare le proprietà delle funzionalità rilevate prima di inserirle nel sistema in esecuzione.

Questa tesi si pone come obbiettivo di fornire tecniche e strumenti atti a supportare specifiche incomplete nel processo di sviluppo software. La tesi propone un formalismo che supporta nativamente l'incompletezza chiamato Incomplete Büchi Automata (IBAs). Gli IBAs estendono i Büchi Automata (BAs) con degli stati non specificati, chiamati *black box* che incapsulano funzionalità ancora da definire. I black box possono essere ricorsivamente raffinati in altri (Incomplete) Büchi automata. Al fine di analizzare gli IBAs proponiamo una tecnica di verifica basata su automi che consente di verificare se un IBA $\mathcal{M}$ soddisfi una proprietà ϕ, specificata in Linear Temporal Logic (LTL). A causa della presenza degli stati black box, la procedura produce tre differenti valori, "si" se la proprietà è soddisfatta dal modello, "no" (e un controesempio) se non è soddisfatta, "forse" se la validità della proprietà dipende nel raffinamento degli stati black box. In questo ultimo caso, viene proposta una tecnica di sintesi capace di computare un vincolo per le parti del modello non specificate. Il vincolo è un insieme di sottoproprietà che devono essere soddisfatte dagli automi che rimpiazzeranno gli stati black box nel processo di raffinamento. Tali automi vengono chiamati replacements. Le sottoproprietà possono essere utilizzate per guidare lo sviluppatore nel processo di raffinamento. Infine, questa tesi propone una tecnica di verifica capace di considerare il raffinamento associato a un black box in riferimento al corrispondente vincolo. Questa procedura evita la verifica del modello originale ogni volta che un raffinamento è proposto, rendendo l'analisi del modello incrementale.

L'approccio presentato in questa tesi è stato implementato nel framework `CHIA` (CHecker for Incompete Automata)[2]. `CHIA` è un tool in versione prototipale che supporta il progettista nello sviluppo del software e nella sua verifica. Il tool è stato utilizzato per valutare l'approccio su due esempi pratici. Il primo è un esempio classico nel campo dell'informatica e riguarda un sistema di mutua esclusione. Tali sistemi sono stati considerati in molti lavori, per esempio [9, 104]. Il secondo esempio, descritto in [139], si riferisce all'evoluzione di un sistema di spostamento pezzi-"Pick and Place Unit"-all'interno di una catena di montaggio.

Infine, per valutare la scalabilità dell'approccio, sono stati considerati un insieme di modelli randomici di dimensione crescente. È stata analizzata la differenza di prestazioni, in termini di tempo e spazio, tra la verifica del replacement associato a un black box in riferimento al vincolo corrispondente e la verifica del raffinamento (il modello originale nel quale il replacement del black box viene inserito) in riferimento alla proprietà originale.

[2]Il framework è disponibile all'indirizzo `https://github.com/claudiomenghi/CHIA/`.

Contents

CHAPTER 1

Introduction

"What we see depends on mainly what we are looking for."

Sir John Lubbock, 1834-1913

In the last few years, software systems overspread the human society. Software pervades every aspect of the every day life: electronic banking, telephone and medical systems, are only few examples of highly computerized systems which are of common use. The massive development of software systems is continuously supported by the reduction of hardware costs, and in particular memory costs, and the grow of the Internet which allows a constant communication between the software executed on the different devices connected to the network [105].

The software development industry is obviously affected by the dynamism of this technological environment. Developers are no longer talented individual people which write the whole software system in isolation. Conversely, nowadays software is developed by tens or thousands of programmers which interact, share code, ideas and components. Sometime developers do not even reside in the same physical location and use other software systems to communicate and integrate their work. In this setting, software development life-cycles evolve from being purely sequential and monolithic to iterative, incremental and agile [50]. Instead of being obsessed by a complete elicitation of requirements, followed by a waterfall shaped development based on hierarchical teams of highly specialized engineers, in an agile approach requirements and solutions evolve through collaboration between self-organizing, cross-functional teams. These factors, in conjunction with the growing complexity of software systems and their increasing interaction, make these systems more vulnerable to errors and malfunctions [9].

Errors may have devastating consequences. In 1994, the crash of a Royal Air Force (RAF) Chinook helicopter killed twenty-five passengers due to an error in the digital engine control system. In 1983, the warning system of the Soviet Union reported incoming US missiles from bases in United States. Luckily the officer recognized the false alarm and decided to do nothing. To prevent these errors, it is crucial to ensure that the software under development satisfies the properties of interest, i.e., its functional and non functional requirements. Functional requirements concern the effect of operations the system is expected to deliver [135]. Typically, functional requirements specify the behavior of a component, such as "after a message is sent it is finally delivered" and "two processes cannot access a critical section together". Non functional requirements refer to the software characteristics, such as performance, availability, usability, energy consumption, and costs [83, 136]. Examples of non functional requirements are "the probability that a message is not delivered must be lower than 0.001" and "the encryption of the sensitive data must take less than 1 second". A system is said to be correct if it meets its functional and non functional requirements.

Verification concerns a set of activities and practices performed to guarantee that the software under development possesses the properties of interest. One of the most used verification techniques is testing [146]. Testing requires to choose a representative set of input values that provide useful information about the behaviors exhibited by the running software and to check the correctness of the produced outputs. The set of test cases must provide enough evidence to give the developer confidence that the system is providing the desired behavior [54]. This makes testing strongly dependent on the correct selection of the suite of test cases. Another technique to check whether the software possesses the properties of interest is model checking [33]. Model checking is usually performed on a model which abstracts the behavior of the real system and reduces the risk of implementing a flawed design. Given a model of the system $\mathcal{M}$ and a formal property ϕ, the model checking tools exhaustively analyze the state space of $\mathcal{M}$ to check whether all of the system behaviors satisfy ϕ [9]. The property ϕ, usually expressed as a temporal logic formula, specifies the requirements the system must satisfy.

Model checking has matured to a stage where practical use is often possible. It has already been used in practice in several application domains and has been adopted in many industrial settings. However, formal verification techniques are still not fully integrated with the current software development cycles, i.e., they do not support incomplete and evolving specifications.

The development process of any complex system can be viewed as a sequence of *design* decisions that make the system evolve from an initial, high-level model into a fully detailed and verified implementation. Typically, this process is performed by iteratively decomposing the model of the system into smaller functionalities. We would like the initial specification to allow a wide collection of possibly inequivalent implementations, which is constantly reduced during the design process, until one single implementation is determined [77]. Thus, at each stage, the model may be deliberately *incomplete*, either because development of certain functionalities is postponed or because the implementation will be provided by a third party, as in the case of a component-based or a service-based system. In the case of a postponed functionality, an implementation is usually provided at some later stage of the development process,

possibly after exploring alternative solutions to evaluate their trade-offs. There are also cases in which the postponed functionality may become available at *run-time*, as in the case of dynamically adaptive systems. A similar process is performed when the system undergoes future evolution.

In this setting, the benefit of analysis performed using classical verification techniques, such as model checking, only appears at the end of a costly process of constructing a comprehensive behavior model, which contains a fully description of the system. Indeed, verification techniques usually do not support the verification of incomplete models. Conversely, highly explorative iterative and incremental model-driven design approaches, like the one we discussed insofar, require the existing formal verification techniques to be profoundly revisited, i.e., they should accomodate *incomplete* designs. More precisely, classical model checking techniques, which return two possible values depending on whether the requirements are satisfied or not in the current model, must consider the case in which the satisfaction of a property depends on the still to be refined components. In this case, the model checking algorithm must return a *maybe* value which specifies that the satisfaction of the property depends on *how* the incomplete is refined.

Furthermore, when the answer is maybe, the developer may also be interested in knowing the set of models that makes the property (not) satisfied, i.e., having an upper bound on the behaviors of the system which guarantee that the property of interest is satisfied. These behaviors represent guidelines, a *constraint*, the developer may follow in the refinement activity. As changes are made, by either adding a part that was previously not specified or by revisiting a previous design decision, we want to ensure that only a minimal part of the system-the one that is affected by the change-needs to be analyzed, thus avoiding re-verifying everything after any change. This would otherwise become intolerably expensive in practice and would alienate practical interest to incorporating formal verification into agile development processes.

This thesis tries to reduce the gap between verification and modern development processes, where, at each stage, alternative design decisions should be explored, and models should be progressively transformed, along with the required properties, until the code level is reached and all assurances are checked. This thesis proposes a modeling formalism and a model checking tool that support the designer in the development of systems which contain *incomplete* parts which are iteratively removed during the refinement activity. The modeling formalism can be used both during the design phase, when the developer may be uncertain about the refinement of some components of the system, and at run-time, when new components can be plugged or removed from the running system. The model checking framework allows the developer to check the incomplete specification of the system supporting an early error detection. The work also proposes a constraint computation algorithm which gives the developer a set of sub-properties the designer must follow in the refinement of the incompletenesses. Finally, the thesis proposes a model checking algorithm which considers only the refined part against the previously generated constraint.

1.1 Research baseline

The work in this thesis is mainly based on the following assumptions:

- *Incomplete designs are baselines to develop fully specified and detailed models of the systems.* The development process is not a straightforward activity: the design of the system may include parts whose behavior is not clear at the current development stage. The developer usually includes these parts in his/her design, but the corresponding specific implementation will be later discussed. Thus, the model of the system is often incomplete.

  Example 1.1.1. *Let us consider the design of a sending message protocol. The developer may start defining a high level view of the system, with an initial state, which is the state from which the system starts, the success state, which is reached when the system has correctly sent the message, and the abort state, which is reached when the sending message activity fails. Then, he/she defines two states* $send_1$ *and* $send_2$ *that represent two attempts of sending a message. When the first attempt fails the state* $send_1$ *is left and the system moves to the state* $send_2$*. At the current level of abstraction, the developer is not interested in defining precisely the behavior of the system inside the states* $send_1$ *and* $send_2$*; he/she is sketching a solution identifying the main components of the application and how they interact each other.*

- *Software development concerns iteratively refining incomplete designs.* The refinement of an incomplete design is a critical task. The developer may provide different alternatives which may differ both in term of their functional and non-funcional characteristics. He/She may want to explore and compare different design alternatives before finally choose one of them.

  Example 1.1.2. *Considering the sending message protocol previously described, the two states* $send_1$ *and* $send_2$ *may represent two different sending procedures, i.e., they may specify different strategies to send a message, such as the use of different physical links. The developer may define a protocol which in the state* $send_1$ *tries to send a message using the optical fibre, while in the state* $send_2$ *uses a third generation mobile telecommunications technology or vice versa. Whenever a failure in the use of any of these links is detected, the system may try to analyze the state of the connection and, in case of problems, to recover it. This analysis can be performed inside one the states* $send_1$*,* $send_2$*, or in both of them.*

- *The choice of the design to be employed depends on its functional and non-functional properties.* When the developer has to choose the design to be finally implemented, he/she may consider a set of characteristics which include both functional and non-functional aspects.

  Example 1.1.3. *In the sending message protocol example, how can the developer choose between the different refinements? Whenever the sending message activity performed inside the state* $send_1$ *fails, does the system have recover the optical fibre connection? Is it better to start the recovering process only after also the sending message activity performed using the third generation mobile telecommunications technology fails? The developer may choose between these two solutions depending on the functional and non functional requirements of the application. For example, one functional requirement may specify that the recov-*

ering can be performed if and only if both the communication channels are not available.

- *Incompleteness is the cornerstone for adaptability.* A system with one variant may be insufficient in most of the real world problems. In real world problems a system has usually many possible replacements for each component. When a component changes (evolve), its old version is removed from the system (leaving its specification temporary incomplete) and replaced by the new one.

  Example 1.1.4. *Imagine that the sending message protocol previously described is deployed on a real system and the state $send_1$ is associated with different strategies to be used at run-time to send the massage. For example, $send_1$ can be associated to two components that use different network links (fibre connection and WiFi). The adaptation procedure can be framed in a model-driven way; that is, the adaptation procedure is based on the use of a model which is kept alive and updated at run-time. Whenever the fibre connection does not allow the sending message activity, the system chooses to change the communication protocol, for example by exploiting the WiFi connection. In this case, the fibre connection component associated with the state $send_1$ is removed from the model of the system, leaving the system temporarily incomplete and replaced by the WiFi component. The model of the system is checked against the corresponding requirements. If the new configuration guarantees the requirement satisfaction, the WiFi component is plugged into running system, otherwise other configurations are analyzed.*

- *Verification proves the correctness of software under development.* System verification techniques are used to establish if the software posses certain properties. A defect is found when the system does not satisfy one of its properties.

  Example 1.1.5. *Considering the sending message protocol example, the developer may want to ensure that the sending activity succeeds, i.e., after a message is sent, a success state is always reached. If the property is satisfied the design is approved, if it is not the design must be revisited.*

1.2 Questions

Software development is not a trivial activity. It is an incremental and iterative process which requires the elaboration of the model of the system, the evaluation of different design solutions, the integration with new and existing software. *Incompleteness* may appear in several stages of the software development cycle. It may occur when a component still has to be specified, when it is removed from the running system, or when it is going to be developed by third party companies. These are common cases that occur when the system is developed using a top-down approach or in adaptive systems, when new components are plugged and removed at run-time. In all these cases, the developer may be interested in *checking* designs that are incomplete.

Even if incompleteness has been exhaustively studied in the research community, a fully comprehensive approach that supports the designer in the development of a *sequential* systems is still missing. Differently from concurrent systems, where multiple processes are executed in parallel, a sequential system is based on a single process.

The behavior of this process is usually described through a state machine which specifies the set of states reachable during the computation and how the state of the system changes over time. This thesis mainly addresses the following questions:

- Incompleteness may occur is several stages of the software development cycle, such as when the development of a certain component is postponed to a later development stage or when it is developed by a third party company. This raises the question:

  Question 1.2.1. *How can the developer specify systems which contain parts whose refinement is postponed? How is it possible to effectively specify the relation between these parts and the rest of our model?*

- Incompleteness and run-time adaptation are two aspects of the same problem. When a component changes, it is removed from the running system leaving the system specification temporarily incomplete. This incomplete specification evolves into a complete one when the new component is plugged. The relation between incompleteness and evolution raises the following question:

  Question 1.2.2. *Whenever it is possible to isolate the portions of the system that will change at run-time, how is it possible to specify these components? How to update the model of the system when the replacements of these components are available?*

- Model checking gives the developer feedback about its design choices. It is usually performed at the end of the development cycle when the final specification is provided. The presence of incompleteness in the specification induces the following question:

  Question 1.2.3. *Are classical model checking techniques able to support the verification of incomplete specifications? If this is not the case, how is it possible to adapt the existing techniques? Is it possible to distribute in a more effective way the verification effort in the development life-cycle when incomplete specifications are refined?*

- Model checking can be used at run-time when new components are detected. In this case, the system has to check whether the plugging of the components make the requirements of the system satisfied or not. If the requirements of the system are satisfied, the component can be plugged into the running system. If this is not the case the components are discarded. This raises the following question:

  Question 1.2.4. *Is it possible to not verify everything from scratch when new components (replacements) are detected? Is there a way to move some of the verification effort from run-time to design-time?*

- When the developer is refining an incomplete part of the system, he/she is designing a replacement, he/she may be uncertain on the behaviors that violate/satisfy the requirements of the system. He/She may need, for example, to choose between two different replacements which differ in their functionalities. This raises the following question:

Question 1.2.5. *Is it possible to compute a constraint (a set of requirements) that the developer must consider in the refinement of incompleteness? How to compute this constraint? Whenever a replacement is proposed is it necessary to check everything from scratch?*

1.3 Contribution of the thesis

This thesis proposes a framework that supports incompleteness in the design of sequential systems, where the systems are expressed in terms of state machines. The contributions of this thesis include:

- *A modeling formalism to represent incomplete models*. Whenever a new system is developed, the designer starts by identifying the set of the system states and how the state of the system changes over time. However, in several cases, the developer may be uncertain about the behavior of the system inside some of its states and can mark these states as placeholders for other state machines that will be later developed. Several modeling formalisms have been proposed in literature to overcome this problem, such as Statecharts [62], Hierarchical State Machines [6] and Modal Transition Systems [77]. Some of these formalisms, such as Statecharts and Hierarchical State Machines, allow the iterative development of the model of the system but do not support early analysis, i.e., the model can be checked against its requirements only when the final, fully comprehensive specification of the system is produced. Other modeling formalisms consider other types of incompleteness. For example, Modal Transition Systems express uncertainty over the presence of a set transitions.

  This thesis proposes Incomplete Büchi Automata an extension of the well known Büchi automata which allow easy identification and specification of incomplete parts. The incomplete parts are represented through a set of states that will be refined into other state machines. Each of these state machines, a *replacement*, is designed in a later development step or, in the case of adaptive systems, can be identified at run-time. Incomplete parts can also be used to abstract parts of the state space, by hiding complex design parts. The thesis provides a formal definition of IBAs, of replacements, and on how IBAs can be iteratively refined adding new replacements.

- *Reasoning techniques*. We develop three reasoning techniques over the modeling formalism previously discussed to support the designer in the development activity.

  - *Incomplete model checking*: allows the developer to check incomplete designs. More precisely, it verifies whether an incomplete model satisfies, possibly satisfies or does not satisfy its requirements. If an incomplete design satisfies or does not satisfy its requirements, whatever replacement for the incomplete parts the developer proposes, the requirement remains satisfied or not satisfied, respectively. If the requirement is possibly satisfied, the developer must refine incompleteness in a way that satisfy the requirements of interest. The incomplete model checking technique is fully automatic, allows an early detection of errors and does not require any user interaction.

  - *Constraint synthesizer*: supports the developer in the cases in which the requirements are possibly satisfied. It computes a constraint, a set of sub-properties, describing the set of replacements that satisfy, possibly satisfy or do not satisfy the properties of interest. The developer may use these sub-properties as guidelines in the replacement design.
  - *Replacement checker*: whenever a new replacement is designed, the developer may want to check whether the new refinement obtained by injecting the replacement into the original model satisfies, does not satisfy or possibly satisfies the properties of interest. The replacement checker does not check everything from scratch, but just considers the replacement against the previously generated constraint.

- *Evaluation and Tool support*. The presented approach has been evaluated over two case studies: the first [9, 104] concerns a well known example coming from academy, while the second is a real case study presented in [139]. We have also analyzed the complexity of all the algorithms proposed in our framework, and the scalability of the approach considering a set of randomly generated models. The entire approach has been implemented in the `CHIA` (CHecker for Incompete Automata) framework[1]. `CHIA` is a prototype tool which supports the designer in the system development and its verification.

1.4 Structure of the thesis

The thesis is structured as follows:

- *Chapter 2* contains an overview of the state of the art. The state of the art includes: *a*) the modeling formalisms that can be used to specify incompleteness in the model of the system to be verified; *b*) how the different modeling formalisms support the refinement of incompleteness; *c*) the techniques that can be used to check incomplete designs; *d*) the algorithms that provide the developer feedback on how to design the replacements of the unspecified components; *e*) the approaches that allow reducing the verification effort necessary to check the automata obtained after the refinement of an incompleteness.

- *Chapter 3* briefly recalls the main steps of the classical automata based model checking procedure. It describes Büchi automata (BAs) and Linear Time Temporal logic (LTL) which are two of the modeling formalisms considered by the commonly used model checking tools. Furthermore, the chapter describes the classical automata based model checking framework, providing the reader the background necessary for understanding the rest of the thesis.

- *Chapter 4* proposes Incomplete Büchi automata (IBAs), an extension of BAs that support incompleteness. The chapter discusses how IBAs can be iteratively refined into other IBAs until the final design of the system is provided.

- *Chapter 5* contains the set of reasoning techniques to analyze incomplete models. The first reasoning technique is based on the automata based model checking

[1] The tool is available at `https://github.com/claudiomenghi/CHIA`.

framework and addresses the problem of detecting whether the model of the system satisfies, possibly satisfies or does not satisfies its requirements. The second reasoning technique synthesizes a constraint that the replacements of the incomplete parts must satisfy. The third reasoning technique allows the distribution of the model checking overhead in a more uniform way over the different stages of the software development cycle. More precisely, when a replacement is provided it is only checked against the previously computed constraint, i.e., it is not necessary to check from scratch the refined version of the model against the original requirements.

- *Chapter 6* describes `CHIA` (CHecker for Incompete Automata) the framework that supports the specification of incomplete models and their refinements and the reasoning techniques proposed in Chapter 5. The chapter describes the architecture of the tool, its input formats and a systematic process to use the tool in the software development.

- *Chapter 7* applies the modeling and reasoning framework on two different case studies. The first is a well known example coming from academy [9, 104]. The second is a real case study presented in [139]. The case studies have been slightly modified to fit with the modeling formalism presented in Chapter 4. The models are transformed since the approach presented in this thesis only supports *sequential* systems, i.e., it does not support parallel execution, and in BAs transitions rather than states are labeled.

- *Chapter 8* analyzes the applicability of the approach. It discusses the performance of the tool when applied to the case studies presented in Chapter 7. The time complexity analysis of the approach is presented and the scalability evaluation is performed by considering a set of random models with increasing sizes. Finally, the chapter discusses the generality of our approach with respect to other approaches that consider incomplete specifications.

- *Chapter 9* concludes the thesis and presents some future work. We outline a set of limitations of the proposed approach and possible extensions.

CHAPTER 2

Related work

"If we knew what it was we were doing, it would not be called research, would it?"

Albert Einstein, 1879-1955

The term incompleteness is generally used to indicate not fully specified systems or systems which do not have all the necessary or appropriate parts. In the software case, incompleteness arises in different stages of the software development cycle and for different reasons. For example, incompleteness may arise in the early stages of the software development when an initial draft (approximation) of the software is designed [130]. This preliminary draft may contain parts that are not specified, i.e., it is a partial design. Abstraction [33] is another technique that generates incomplete specifications. Abstraction allows hiding details of the system to reduce the size of the model. The abstracted models are designed to be conservatively true with respect to the properties of interest, i.e., if a property is satisfied in the abstract model it is also satisfied in its refinement.

Whenever an incomplete model is considered, the developer may clash with the following problems: *a*) how to specify models and claims which are incomplete; *b*) how to refine the incomplete models; *c*) how to check the incomplete models and claims; *d*) how to synthesize constraints for the incomplete parts, i.e., sub-properties the developer may consider in the refinement activity; *e*) when an incomplete part is refined, how to check the new model of the system. This chapter reviews the main works and solutions proposed for these problems. The focus of the chapter is on the possible ways in which incomplete models are specified and analyzed and does not consider incompleteness in the property (claim) to be verified. Incompleteness in the properties to be checked has been strongly studied in literature. The problem of checking incomplete

properties is identified in literature as query checking [20]. The interested reader may refer to [60] for additional information.

2.1 Modeling formalism

Different modeling formalisms have been proposed in literature to support incompleteness. Most of them are variations of the ordinary *Finite State Machines* (FSMs). In FSMs a system is usually modeled through a set of states and transitions between these states. Transitions specify how the configuration of the system changes over time.

Modal Transition Systems (MTSs) are one of the most studied modeling formalism to deal with incompleteness. MTSs have been originally proposed by Larsen and Thomsen [77, 78] as an extension of Labeled Transition Systems (LTSs). LTSs are not suitable to describe initial, high-level specifications where a wide collection of possibly not equivalent models can be defined and then refined during the development process until a single (final) implementation is determined. MTSs extend the formalism of LTSs by dividing the transitions of the LTSs into necessary and admissible[1]. A transition $s \xrightarrow{a} s'$, which specifies that the system moves from the state s to the state s' by reading a, may be prefixed by a *necessary* ($\Box$) or an *admissible* ($\Diamond$) operator. A transition prefixed by a necessary operator ($\Box(s \xrightarrow{a} s')$) specifies that the system *must* be able to perform the transition. A transition prefixed by the admissible operator ($\Diamond(s \xrightarrow{a} s')$) specifies that the transition *may* be implemented by the system. Formally, a MTS $\mathcal{M}$ is a tuple $\langle S, A, \rightarrow_{\Box}, \rightarrow_{\Diamond} \rangle$, where S is the set of the states of the MTS, A is the set of actions, $\rightarrow_{\Box} \subseteq S \times A \times S$ is the set of necessary transitions and $\rightarrow_{\Diamond} \subseteq S \times A \times S$ is the set of admissible transitions, such that $\rightarrow_{\Box} \subseteq \rightarrow_{\Diamond}$[2].

Kripke MTSs [67, 123] (KMTSs) are similar to MTSs. They represent the *necessary* and the *possible* behaviors of a system by partitioning the set of transitions in two sets: must ($\overset{must}{\rightarrow}$) and maybe ($\overset{maybe}{\rightarrow}$) transitions. The set of must transitions is included into the set of maybe transitions. However, an abstract state s_a of the KMTS $\mathcal{M}$ is used to represents a set of concrete states of the Kripke Structure (KS) $\mathcal{M}'$ which refines $\mathcal{M}$. Furthermore, differently from MTSs, states rather than transitions are labeled. The labeling of an abstract state contains the atomic propositions that are satisfied or not satisfied by all the concrete states of $\mathcal{M}'$. However, the value of some of the proposition of a state s may not be specified in $\mathcal{M}$ and assigned only when s is refined.

Generalized Kripke MTSs [123] (GKMTSs) replace *must* transition with *must hyper-transitions* which connect an abstract state s_a with a set of states $A = \{s_1, s_2 \dots s_n\}$. A GKMTS contains a transition $s_a \overset{must}{\rightarrow} A$ if and only if for each concrete state s_c represented by s_a there exists a s'_c represented by some $s'_a \in A$ such that $s_c \rightarrow s'_c$. In GKMTS an abstract state s_a can be mapped into several concreted states s_c.

Partial Kripke Structures [11, 12] (PKSs) are a state based modeling formalism which allow the specification of incomplete models. Given a set of proposition P, a partial Kripke Structure is represented as a tuple $\langle S, L, R \rangle$, where S is the set of states, $L : S \times P \rightarrow \{T, \bot, F\}$ is the interpretation function and $R \subseteq S \times S$ is a transition relation on S. As evidenced by the interpretation function L, PKSs associate to

[1] These transitions are also identified as *possible* transitions, such as in [21].

[2] The interested reader may refer to [130] for a review on the current state of the art of MTSs, and how they are used to support incremental model elaboration.

each proposition a truth value in $\{T, \perp, F\}$ for each state in S, i.e., a proposition can take also the truth value $\perp$. This value is used to specify propositions whose value is currently undefined, it can be either true or false[3]. A PKS can be used in the model checking activity to iteratively explore the state space of the system. The set of states still to be explored by the checker can be represented as a single state $s_\perp$ that models all of them and has the value $\perp$ assigned to all the atomic propositions. Thus, while it is operating, the model checker explores the state space and maintains an abstracted version of the real state space of the system where the states still to be visited are represented by the single abstract state $s_\perp$. Alternatively, states where all the propositions are assigned to the $\perp$ value can be used in the software development as abstract states that have still to be refined.

$\mathcal{X}$Kripke Structures ($\mathcal{X}$KSs) [22] extend KSs by labeling states and transitions with multivalued sets. A multivalued set is a boolean algebra where a variable can be marked with TT, FT, TF and FF. Intuitively, an atomic proposition a can be associated to a value in the set $\{TT, FT, TF, FF\}$. When the value of the proposition a is FT or TF, its actual value is unknown. Similarly, every transition is associated with a value in the set $\{TT, FT, TF, FF\}$ which specifies whether the transition is present, not present or possibly present in the final KS. Again, when a transition δ is marked with FT or TF its presence in the final model is not guaranteed.

Hierarchical State Machines [6] (HSMs) are usually considered as a notation to deal with incomplete specifications. In HSMs states can be ordinary states or superstates (also called boxes) which are refined into other HSMs. Each ordinary state is labeled with the atomic propositions true in that state. Furthermore, a HSM has one state denoted as entry state and one or more ordinary states denoted as exit states. Entry and exit states allow to connect the HSM of the current level with a HSM of the lower/upper levels. HSMs offer two main advantages: *a*) superstates allow the specification of systems in a stepwise refinement way and to consider the system at different levels of granularity; *b*) it is possible to map different superstates on the same HSM, i.e., it is possible to specify a component only once and plug it into different superstates. In this case, HSMs allow *sharing*. Formally, a HSM is composed by a set of structures, i.e., HSMs, which are connected by an indexing function which maps superstates to other HSMs. The transition relation (called edge relation) connects together states of the same HSMs or states of a HSM with entry/exit nodes of another HSM.

Giannakopoulou et al., [56] consider LTSs as a formalism to express the behavior of the system when it is executed in an unknown environment. The environment can interact with the LTS by triggering some of the actions which label the transitions of the LTS. The set $\mathcal{A}$ of the actions of the model which are observable from the environment are specified through the *interface operator* $\uparrow$. In the rest of this thesis this type of LTS will be indicated as LTS$^\uparrow$. The interface operator intrinsically makes the model incomplete, meaning that some actions may be triggered or not by the environment. From the environment point of view, all the actions which are not in $\mathcal{A}$ can be replaced by the symbol τ which is outside the alphabet, since they are not observable. More precisely, the transitions marked with the τ symbol cannot be executed synchronously with transitions of the claims or of the environment, but instead they are interleaved.

In [95], incompleteness of sequential circuits is considered. In this case, incom-

[3]Note that, in general, the symbol $\perp$ can be associated with different interpretations [31].

pleteness is used to describe a specific part of the circuit whose behavior is unspecified, i.e., we know its inputs and outputs but not the internal representation of the logic. The output of the system is marked with an X meaning that its value is unknown (0 or 1) for a specific input. In this case incomplete parts are named as black boxes.

In other works, such as [29, 30], the system is decomposed into a set of unknown components which communicate through signals. The developer knows the structure of the system, i.e., how the different components are interconnected, and wants to produce contracts for these components. The components are represented as black-boxes, while their behavioral models are specified in a different language. Each component has a set of incoming and outgoing ports. The incoming ports receive signals from the environment, i.e., the surrounding components. The outgoing ports are used to communicate signals to the other components of the system.

A concept which is often associated with incompleteness is uncertainty. The simplest example of uncertainty is non-determinism, which has been considered for example in [28]. In a non-deterministic domain a model contains actions which have different outcomes and whose occurrence cannot be predicted at design time, i.e., it is not possible to know a priori the set of action (e.g., transitions) that will be executed at run-time. When the model is non deterministic, an execution may result in general in different sequences of states.

Uncertainty has also been considered in [44]. In this work multiple design possibilities are associated with a component, i.e., uncertainty refers to “multiple design alternatives". The developer proposes different design alternatives for the same component but he/she is uncertain on the one to select and deploy in the final system.

Several works also consider incompleteness when the model is used to represent the non-functional properties of the system [45]. In this setting incompleteness refers to the values of the parameters of its transitions, e.g., the probability to move from one state to another.

2.2 Refining process

The refinement process is an iterative activity in incompleteness is iteratively removed. The refinement operation is the base block of the software development cycle, in particular when a stepwise development technique is used. This section discusses the refinement notions of some of the modeling formalisms that support incompleteness.

Different definitions of refinements have been proposed for *Modal Transition Systems* (MTSs) [48, 49, 77, 131, 132][4]. The common idea behind these refinement notions is to refine a MTS by converting maybe transitions into required or proscribed one. The first notion of refinement, also known as classical refinement or strong refinement, is described in the original work of MTSs [77] and considers two MTSs defined over the same alphabet. A MTS $\mathcal{M}'$ is a refinement of a MTS $\mathcal{M}$ ($\mathcal{M}' \lhd \mathcal{M}$[5]) if every admissible behavior in $\mathcal{M}'$ is also an admissible behavior in $\mathcal{M}$, and every necessary behavior in $\mathcal{M}$ is necessary in $\mathcal{M}'$. A MTS $\mathcal{M}$ is called implementation if $\rightarrow_{\Box} = \rightarrow_{\diamond}$, that is, $\mathcal{M}$ is a labeled transition system [107]. Note that the refinement relation $\lhd$ is a pre-order (reflexive and transitive) and is a generalization of the notion of bisimulation [90, 97].

[4] The interested reader may refer to [21, 130] for additional information on the use of MTSs in the context of incremental behavioral model elaboration.

[5] The classical refinement is also indicated in some works with the symbol $\preceq$, such as in [21].

When $\rightarrow_{\Box} = \emptyset$, the refinement relation $\lhd$ corresponds to the notion of simulation, while corresponds to bisimulation when $\rightarrow_{\Box} = \rightarrow_{\diamond}$. The classical refinement notion has been extended in [48, 131] to accomodate different alphabets, while in [21, 46, 47] observational refinement (usually indicated with $\preceq_o$) is introduced. Observational refinement is useful when the alphabet Σ of a MTS $\mathcal{M}$ is extended with the additional symbols, such as τ, which describe actions of $\mathcal{M}$ which are not visible by the MTSs running in parallel with $\mathcal{M}$. In this case, the alphabet of the MTS is $\Sigma' = \Sigma \cup \{\tau\}$ which is composed by the communicating part Σ and the non observable one τ. The observational refinement reflects the classical refinement notion considering only the communicating alphabet of the MTS. Given a MTS $\mathcal{M}$ and a set of symbols X, the MTS $\mathcal{M}@X$ is the MTS $\mathcal{M}$ where all the occurrences of a symbol in X is replaced with τ. Observational refinement allows the comparison of a model with a biggest one by hiding the additional actions present in the biggest model.

The refinement notion of *Partial Kripke Structures* [11,12] (PKSs) is defined through the completeness pre-order relation $\preceq$, which relates properties of less complete PKSs to more complete PKSs. Given a model $\mathcal{M}$ and another model $\mathcal{N}$, $\mathcal{N}$ is a refinement of $\mathcal{M}$ if there exists a completeness pre-order relation between their states. Given two states s_1 and s_2 of $\mathcal{M}$ and $\mathcal{N}$, respectively, $s_1 \preceq s_2$ if the atomic propositions of s_1 are "less defined" than the atomic propositions of s_2 and the successors of s_1 and s_2 are in a pre-order correspondence. In other words, a model $\mathcal{M}$ and another model $\mathcal{N}$ are in a pre-order relation if $\mathcal{N}$ is "more complete" than $\mathcal{M}$, which means that $\mathcal{N}$ has "more" definite properties with respect to $\mathcal{M}$. The completeness pre-order relation can be used both for abstracting (P)KS by encapsulating portions of the state space into abstract states or to iteratively refining incomplete specifications.

The refinement process of *Hierarchical State Machines* (HSMs) is obtained by connecting a *superstate* of a HSM to another HSM (its replacement). When the refinement process is ended, a HSM can be converted into a flat FSM K^F, usually called *expanded* FSM (or expanded structure), by recursively substituting each box of the structure by the corresponding FSM [8]. As demonstrated in [5] such flattening causes an exponential blow up, i.e., $\mathcal{O}(|K|^{nd(K)})$ where $|K|$ is the size of the HSM and $nd(K)$ is the *nesting* depth, i.e., the the length of the longest "refinement chain".

Giannakopoulou et al., [56] assume the model of the system specified through a Labeled Transition System with an additional *interface operator* (LTS$^{\uparrow}$) , which describes how the model can interact with its environment. The refinement activity concerns the specification of the environment in which the model is executed.

In [29, 30] a component (black-box) can be refined into other components. The developer starts with a view of the system which is composed by a single black box with a set of ports that allow to communicate with the environment. Then, the black-boxes are iteratively refined. At each refinement round, the developer specifies the sub-components included in the refined component. Additionally, the developer also specifies how the incoming and outgoing ports of the sub-components are connected together and to the incoming and outgoing ports of the refined component.

2.3 Model Checking

Several works have considered the problem of checking models that are incomplete. When an incomplete model is analyzed, the output of classical model checking techniques must be revisited. Classical model checking algorithms, given a model of the system $\mathcal{M}$ and a property ϕ, return "true" if the property is satisfied, "false" if it is not. Model checkers that support incompleteness must be *multivalued* [22], i.e., they should rely upon some type of three value [71] or multivalued logic.

A multivalued model checking framework allows the reasoning on additional truth values rather than just "true" or false". A particular instance of multivalued model checking is *3-valued* model checking [11] where three possible outputs can be returned by the model checker: "true", "false" and "maybe" (or possibly). The procedures proposed in literature differ from the modeling formalism considered by the model checking framework and on the type of result the tool is going to provide. This section mainly considers the state of the art with respect to the verification of LTL properties.

The model checking problem of *Modal Transitions Systems* (MTSs) has been considered for example in [66, 67, 129][6]. The model checking procedure is based on two classical model checking activities. First, an under approximation $\mathcal{M}^-$ of the model $\mathcal{M}$ to be checked is considered. To obtain $\mathcal{M}^-$ all the maybe transitions are removed from $\mathcal{M}$. Then, all the required transitions that are not part of an infinite run that starts from the initial state of the system are deleted. The new automaton is checked against the property of interest. If $\mathcal{M}^-$ does not satisfy ϕ the original model $\mathcal{M}$ does not satisfy the property of interest. If this is not the case the over approximation $\mathcal{M}^+$ is computed. $\mathcal{M}^+$ is obtained by converting all the maybe transitions into required and by removing the transitions that are not a part of infinite runs starting from the initial state of the system. If $\mathcal{M}^+$ satisfies ϕ, then ϕ is satisfied in the final model, otherwise, the property is possibly satisfied. As a consequence the verification procedure has a complexity which is comparable to the classical model checking algorithms, i.e., the overall complexity is $\mathcal{O}(2 \cdot |\mathcal{A}_{\neg\phi}| \cdot |\mathcal{M}|)$, where $|\mathcal{A}_{\neg\phi}|$ is the size of the automaton associated with the property $\neg\phi$ while $|\mathcal{M}|$ is the size of the model to be checked, respectively.

The model checking of $\mathcal{X}$*Kripke Structures* ($\mathcal{X}$KSs) has been considered both with respect to CTL ($\mathcal{X}$CTL) [22] and LTL ($\mathcal{X}$LTL) properties. In particular, checking a $\mathcal{X}$KS structure versus an $\mathcal{X}$LTL formula can be performed by checking $\mathcal{X}$LTL against a $\mathcal{X}$*Büchi automata* [23]. $\mathcal{X}$Büchi automata assign to each symbol a that labels a transition δ its value from a multivalued set. As in the case of Modal Transition Systems the model checking problem can be reduced to a set of executions of the classical model checking algorithm. The overall complexity of the procedure is $\mathcal{O}(2 \cdot |\mathcal{A}_{\neg\phi}| \cdot |\mathcal{M}|)$.

The model checking problem of *Partial Kripke Structures* (PKSs) has been considered in [11]. The main issue of checking a partial state space is to relate the verification of the properties on the partial state space to the results obtained when the procedure is applied to the full state space. In [11], Propositional Modal Logic (PML) [137] and Computation Tree Logic (CTL) [32] is considered. PML extends propositional logic with the eventually operator F and is a subset of both CTL and LTL. The third value $\perp$ is used to specify an "unknown" result, meaning that the model does not contain enough information to say whether the property is true or false. The authors demonstrate that

[6] The model checking problem is considered with respect to the *inductive* semantic of LTL and only deadlock-free MTS are supported. The complexity of the model checking procedure corresponds to the one presented in [11].

given a Partial Kripke Structure $M = \langle Q, L, \Delta \rangle$ it is possible to solve the model checking problem by performing two "normal" model checking activities, one considering an optimistic labeling L_a, where the uncertain atomic propositions are associated with the *true* value, and the other considering a pessimistic labeling L_p, where the uncertain atomic propositions are associated with the *false* value. The formula is true in a state s if it is true in its under pessimistic interpretation, it is false if it is not satisfied in its optimistic interpretation, $\perp$ otherwise. For this reason, the model checking procedure does not have additional complexity: the overall complexity is $\mathcal{O}(2 \cdot |\mathcal{A}_{\neg\phi}| \cdot |\mathcal{M}|)$, where $|\mathcal{A}_{\neg\phi}|$ is the size of the automaton associated with the property $\neg\phi$ while $|\mathcal{M}|$ is the size of the model to be considered.

In [12] the approach previously presented is extended and the *generalized model checking* problem is defined. The authors specify that the 3-value semantics does not behaves as expected: there are cases in which the model checking algorithm returns $\perp$ in which it does not exist a "more complete" model in which the formula is satisfied and not satisfied. For this reason the generalized model checking problem has been defined. The generalized model checking algorithm returns the value unknown $\perp$ if and only if the property is possibly satisfied, but ensures that there exists two refinements one which satisfies and the other which does not satisfy the property of interest. This 3-value semantic is indicated as *thorough* semantic. The generalized model checking problem is therefore a generalization of *a*) the model checking and *b*) the satisfiability problem . If the given structure is fully incomplete, the problem reduces to satisfiability checking while if the given structure is fully complete, the problem reduces to model checking. The generalized model checking procedure over LTL formulae can be decided in time $\mathcal{O}(|S|^2 \cdot 2^{2 \cdot |\phi|})$.

Model checking of *Hierarchical State Machines* (HSMs) with respect to LTL properties has been considered in several works such as [6–8]. However, in all these works, HSMs are not considered as a formalism to model incompleteness, and thus the verification is assumed to be performed at the end of the development cycle when the final implementation of the model is provided, i.e., all the components of the HSM are specified. The verification of a fully specified HSM is discussed in detail in Section 2.5.

Giannakopoulou et al., [56] assume the model of the system specified through a Labeled Transition System with an additional *interface operator* (LTS$^{\uparrow}$). The interface operator describes how the model can interact with its environment. The system obtained by combining the model and the environment is considered against a safety claim also specified in terms of an automaton (a deterministic LTS). The authors revisit the traditional approach that allows verifying a model behavior in all of its possible environments. Indeed, classical approaches return *a*) true if the model (component) satisfies the property in all the possible environments; *b*) false if there exists some environment that lead the component to falsify the property. The authors state that this approach is overly pessimistic since it assume a not helpful environment. Thus, they modify the model checking framework to return the value *a*) true if the model (component) satisfies the property in all the possible environments; *b*) false if the component violate the property in all the environments; *c*) maybe otherwise. In the maybe case it may exists some helpful environment which executed in parallel with the model guarantees that the property is satisfied. The model checking algorithm works in the following steps *a*) it first negates the LTS which corresponds to the property, that is it generates the LTS

$\overline{\mathcal{S}}$, which corresponds to the LTS $\mathcal{S}$ augmented with an error state[7]; *b*) it computes the parallel execution of $\mathcal{M}$ and $\overline{\mathcal{S}}$ ($\mathcal{M} \parallel \overline{\mathcal{S}}$); *c*) it hides the actions that cannot be controlled by the environment into τ actions; *d*) it checks if an error state is reachable in $\mathcal{M} \parallel \overline{\mathcal{S}}$. If it is not reachable the answer is true, if it is reachable through transition marked only with τ actions it is false, otherwise it depends on the behavior of the environment. The complexity of the model checking procedure is linear with respect to the intersection automaton $\mathcal{I}$ generated.

The verification of incomplete *circuits* versus CTL properties is considered for example in [95]. In this case, the model checking framework is split in two parts, realizability check (which is undecidable in general [108]), which verifies if it is possible to refine the incomplete parts in a way that satisfy the original property, and validity check, which verifies whether the property is satisfied in all the possible refinements of the incomplete parts. The results of the checking are an approximation: they are not complete but they are sound. The model checking framework first encodes the incomplete circuit into a symbolic representation, where an additional variable Z_i is added for each Black Box output (i.e., for each output of the incomplete parts). Then it computes an over-approximation $Sat_E(\phi)$ and an under-approximation $Sat_A(\phi)$ of the states of the system which specify whether there exists at least one replacement (or all the replacements) of the black boxes that satisfy ϕ. The sets $Sat_E(\phi)$ and $Sat_A(\phi)$ are computed based on an approximate transition relation.

In the context of very-large-scale integration (VLSI), the problem of checking incomplete designs is considered, for example, in [96]. More precisely, in this work, two sub-problems are analyzed: *a*) realizability, that verify if an incomplete design can be extended to a complete design satisfying a given CTL formula and *b*) validity, i.e., checking whether the property is satisfied for all possible extensions . In this work, incompleteness refers to sub-circuits, that is an incomplete part represents a sub-circuit that can be plugged inside the system. The authors proposed an approximate solution to the realizability problem, since this problem is undecidable in general (a black box can be replaced with a circuit with an unrestricted amount of memory).

2.4 Computation of sub-properties

The main goal of the software development process is to produce a system which satisfies the properties of interest. As seen in Section 2.3, the developer usually designs a model of the system and then check whether it satisfies its properties. However, the reasoning techniques needed by the developer do not only involve model checking. Sometimes the developer wants to “compute" or “synthesize" portions of the system under development. For this reason, problems such as “synthesis", “supervisory control" and “sub-module construction" are related to this thesis. As specified in [56] “the particular framework in which these problems are considered makes all the difference to the proposed solutions".

In *program synthesis* the developer wants to compute a model of the system that satisfies the properties of interest. Depending on the type of the system the developer wants to synthesize and on the information that trigger the synthesis, different procedures have been proposed. For example, the developer may want to synthesize the

[7] Note that the complexity of this procedure is in the worst case linear since it corresponds to the negation of a FSA and not of a BA.

model from the properties the system has to satisfy or from a set of scenarios [72, 134] or from both of them [129].

In [112] the program synthesis problem is considered in the case of *reactive systems*. A reactive system is a system that has an outgoing interaction with its environment, i.e., the model $\phi(x, y)$ characterizes the relation between the input x of the program (obtained from the environment) and the output y. In this context we can consider two different components: the component C_1, which is the environment where the application will be deployed, and the component C_2, which is the body of the system the developer has to design. The point is to show that the developer can design C_2 by employing a winning strategy that for all possible x scenarios makes the property $\phi(x, y)$ satisfied. The system C_2 may react to the inputs of the environment in a way that satisfies the properties of interest. Formally, $(\forall x)(\exists y)\phi(x, y)$, for all input x there exists an action y the component C_2 can produce that makes the final property $\phi(x, y)$ satisfied. The input x and the output y are sequences of values assumed by the variable x and y along the computation. The theorem proving approach, by proving the validity of $(\forall x)(\exists y)\phi(x, y)$, is used to demonstrate the existence of a program P satisfying the linear time temporal formula $\phi(x, y)$. The behavior of a program P is an infinite sequence of pairs $\langle x_i, y_i\rangle$ such that $y_i = f_P(x_i, x_{i-1} \dots x_1)$ where $x_i, x_{i-1} \dots x_1$ are the inputs received by the system and f_P is the program to be constructed. The output y_i of the system in a particular instant depends on the current input x_i and the sequence of inputs $x_{i-1} \dots x_1$ the system has received before x_i. The proof that demonstrates the validity of $(\forall x)(\exists y)\phi(x, y)$ is used to construct a program that implements ϕ (a similar idea has been exploited for example in [15, 42, 140]). In the case of finite systems, the formula generated can be mapped on a propositional CTL formula, whose implementability can be checked and a transducer whose size is at most double-exponential in the length of ϕ can be computed.

In [85] the authors analyze the problem of the synthesis of the *synchronization part* of communicating processes. The processes are specified in the CSP (Communicating Sequential Processes) language which allows the specification of processes interacting with I/O operations. The incomplete part to be synthesized is the synchronizer which filters all the interactions of the communicating processes. The goal of the synthesis procedure is to compute the synchronizer such that the global system, composed by the communicating processes and the synchronizer, satisfies the LTL property of interest. The synthesis procedure works through four steps. First, the relativization procedure transforms the local specifications of the processes into global specifications. This step is necessary since each process is specified separately when it is designed, but when the properties of the system are analyzed (i.e., absence of deadlock), the combination of the specifications of the components of the system must be considered. Second, a tableau like satisfability algorithm for the LTL properties is executed. The output of the algorithm either declares that the specification is unsatifiable, which means that there is no program that can satisfy the synchronization procedure, or it produces a model graph from which all the other possible models can be extracted. In the third step the set of models that satisfy the specifications (unwind the graph to satisfy eventualities if necessary) is filtered. Finally, the synchronizer is generated.

When FSAs are considered [101], the system has to react to the inputs provided by the environment in a way that satisfies its specification described through the LTL

formula ϕ. The goal is to synthesize a FSA $\langle Q, i, \Sigma, \Gamma, \delta, \rho \rangle$ where Q and i is the set of the states and the initial state of the system, Σ and Γ are the input and output alphabets, respectively, $\delta : S \times \Sigma \rightarrow S$ is the transition function and $\rho : S \times \Sigma \rightarrow \Gamma$ is the output function. As in classical synthesis it is not always true that the specification is realizable [110], i.e., the system has a strategy to satisfy the specification. A typical case in which it is not possible to synthesize the system is when the choice of the system depends on some future input. The synthesis algorithm is build upon three main steps *a*) translation of the LTL formula into a Büchi automaton (BA); *b*) determinization of the BA into a Rabin [117] or parity [106] automaton; *c*) search for a winning strategy in the generated automaton. The translation of the LTL formula into a BA and the determinization step have both an exponential blow up. Thus, the overall procedure is doubly-exponential in the size of the LTL specification ϕ.

In the context of FSMs, Uchitel et al., [129] propose a synthesis technique that constructs *Modal Transition Systems* (MTSs) from a combination of *safety properties* and *scenarios*. The idea is that safety properties are used to synthesize a model that represents an upper bound on the behaviors of the system, i.e., they include all the possible behaviors the system can exhibit, while scenarios are lower bounds on the behavior of the system, i.e., they describe less behavior than what the final system shall provide. More precisely in [129], *a*) the safety property is expressed in FLTL [55]. This property is converted into a MTS using a novel algorithm which is based on the classical algorithm [80] used to translate FLTL properties into LTS[8] and on the three value semantic of FLTL[9]. The idea is to obtain the LTS which corresponds to the safety property. Then, the LTS is converted into a MTS: each state of the obtained LTS is analyzed and its outgoing transitions are converted into maybe transitions when more than one transition exit a state since not all of them are required in the refined MTS. *b*) the scenario σ is used to generate a MTS $\mathcal{M}(\sigma)$ which includes all the traces the system must exhibit. Starting from the LTS $\mathcal{L}(\sigma)$ generated from a scenario σ, the MTS $\mathcal{M}(\sigma)$ is obtained from $\mathcal{L}(\sigma)$ by adding a new *sink* state. Each state s in $\mathcal{L}(\sigma)$ is connected to the sink state through a transition labeled with a, if it does not exists a transition labeled with a that exit the state s, i.e., all the traces which are not explicitly described in $\mathcal{M}(\sigma)$ are turned into maybe traces. *c*) after the MTSs generated from the property and the scenario are computed, by *merging* these two MTSs[10] the MTS which represents their least common refinement is created. The new MTS preserves the required (by scenarios) and the proscribed (by properties) behaviors. *d*) by analyzing this new MTS it is possible to check whether the property is satisfied, not satisfied or possibly satisfied. Since the procedure relies on the LTL tableau construction [81] the complexity of the procedure is exponential. The synthesis and the model checking approach have been implemented in the Modal Transition System Analyser [41] tool which is based on the Labeled Transition System Analyzer [84] tool. Other related works on the synthesis of MTSs are for example [39,40,133]. A more complete survey can be found in [130].

A particular instance of the synthesis problem is the *supervisory control* problem. In

[8] When the property ϕ is a safety property the corresponding automaton has only one accepting state.

[9] In the three value semantic, if a property evaluates to *true*, it is satisfied in all the possible executions of the model. If a property evaluates to *false*, it means that it is not satisfied in all the executions of the model. Finally, if a property evaluates to maybe, there are some executions of the model satisfy and some that do not satisfy the property of interest.

[10] The merging operator is defined over two consistent, deterministic MTSs with the same alphabet.

supervisory control [16,92,113,114], given a model of the system $\mathcal{M}$, such as a Discrete Event System (DES) or an Extended Finite Automata (EFA) [125], the problem is to modify its behavior to guarantee that the system satisfies the properties of interest, i.e., to *synthesize* an appropriate controller. The idea is based on the classical control loop concept. A controller $\mathcal{S}$ uses any available information about the system behavior to continuously adjust the input of the system, i.e., to control the executions of the model. The model $\mathcal{M}$ receives inputs and produces outputs which are in form of events. As soon as the model $\mathcal{M}$ produces some events which are detected by the controller $\mathcal{S}$, the controller $\mathcal{S}$ may disable some input events of $\mathcal{M}$ to guarantee the satisfaction of certain properties. Note that, the set of input events of $\mathcal{M}$ is usually partitioned in two sets, i.e., the controllable and not controllable events. The synthesis algorithms for the supervisory control problem aims at computing a strategy the controller can exploit to reach a certain goal. For example, the supervisor can be represented as a function that, for every state of the model $\mathcal{M}$, disables a subset of actions that can be controlled. The main idea behind supervisory control is to guarantee that the parallel execution $\mathcal{M}||\mathcal{S}$ does not allow/allows to reach a given a set of states E that the system has not/has to reach (which can be obtained from a property ϕ). To reach this goal one possible algorithm is to compute the parallel execution $\mathcal{H}$ between $\mathcal{M}$ and an automaton which represents the property ϕ, remove from $\mathcal{H}$ the states that are not reachable obtaining a new FSM $\mathcal{H}^{\uparrow}$. If $\mathcal{H}^{\uparrow}$ is not reduced to the empty FSM it is the greatest controllable sub-machine of $\mathcal{M}$ that ensures the satisfaction of the property. The algorithm is polynomial in the number of the states of the system.

The supervisory control problem has also been analyzed in the context of *decentralized control* (see for example [116, 144, 145]). In the decentralized supervisory control problem, there are "many" controllers $\mathcal{S}_1, \mathcal{S}_2 \ldots \mathcal{S}_n$ in charge of controlling the system. These controllers may have different (possibly overlapping) set of actions which are monitored and may control different (possibly overlapping) set of inputs of $\mathcal{M}$. In this context a conjunctive or a disjunctive architecture can be employed. In a conjunctive architecture the set of enabled inputs of the model $\mathcal{M}$ is obtained by computing the intersection of the set of events enabled by the controllers, vice-versa in a disjunctive architecture the set of enabled inputs is the union of all the events enabled by the controllers.

The supervisory control problem has also been considered in the context of *Hierarchical Finite State Machines* (HFSMs) [10, 86, 143]. A particular interest is obtained by the HFSMs with sharing, i.e., where sub-components can be reused as replacements for several super-states. Several works solve this problem by translating the HFSM into an ordinary FSM and then using the classical approaches over the resulting FSM. However, as mentioned in [51, 86], the synthesis algorithms for the supervisory control problem are polynomial in the number of the states of the system, and the number of the states of the system grows exponentially with the number of nested sub-systems. For this reason, it is important to exploit the hierarchical structure of the system. The algorithm proposed in [86] according to the structure of the hierarchical state machine computes a collection of supervisors (one for each structure) that are generic, computed only once and can work in different context, i.e., inside the refinement of several states.

In [56] the *assumption generation* problem for Labeled Transition System with an additional *interface operator* (LTS$^{\uparrow}$)-which describes how the model of the system in-

teracts with its environment- is considered. If the model $\mathcal{M}$ possibly satisfies the claim ϕ, specified through an automaton (a *deterministic* LTS), the authors propose an algorithm to compute an assumption, which works in any case there exists an helpful environment which may guarantee the satisfaction of the property of interest. In this sense the procedure is similar to the supervisory control problem. The algorithm considers the intersection automaton (where the not observable actions of the model are marked with τ) and goes through the following steps: *a*) it backtracks the error state over τ transitions, pruning the states where the environment cannot prevent the error state from being entered via τ steps. *b*) it extracts the assumption. The assumption to be satisfied by the environment can be extracted by the intersection automaton in two steps: extracting the negation of the property, by making the intersection LTS deterministic, and negating the property. The bottleneck of the approach is the determinization step which in the worst case is exponential and is performed since the claim to be considered must be specified in terms of deterministic LTSs.

In the following work [34], the authors present a framework to perform assume guarantee reasoning in an incremental and fully automated fashion. While in classical assume guarantee reasoning the assumption are provided by the developer, in this work the authors try to automatically derive the assumption $\mathcal{A}$ from the description of the component $\mathcal{M}$, to guarantee that $\mathcal{M}||\mathcal{M}'$ satisfies a certain property ϕ. The framework checks the component $\mathcal{M}$ against a *safety* property ϕ and generates an assumption $\mathcal{A}$ the component $\mathcal{M}$ must satisfy and $\mathcal{M}'$ must guarantee. The assumption $\mathcal{A}$ is generated through a learning algorithm which exploits queries on $\mathcal{M}$ and the results of the checking of $\mathcal{M}$ against ϕ. The framework works in three steps *a*) an assumption $\mathcal{A}$ is generated for the component $\mathcal{M}$ through the learning algorithm; *b*) the model checker is used to verify that the component $\mathcal{M}$ satisfies the property ϕ under the assumption $\mathcal{A}$, if the property is not satisfied another assumption $\mathcal{A}$ is generated using the learning algorithm and the result of the verification; *c*) the model checker is used to verify that the component $\mathcal{M}'$ satisfies $\mathcal{A}$ in any environment. If this is the case the procedure terminates, otherwise another assumption is generated through the learning algorithm and the result of the verification. The component $\mathcal{M}$, the property ϕ and the assumption $\mathcal{A}$ are described through LTSs. Note that the LTS that describes the property must be *deterministic*[11]. In [34], the L^* learning algorithm is used to generate the assumptions.

Cimatti et al., [30] propose a framework that supports the developer in the refinement process by decomposing contracts when a top-down development process is adopted. Whenever a component is refined into sub-components the corresponding contracts are defined. Contracts specify the expected behavior of components in terms of assumptions the environment must satisfy and guarantees the components provide in response, i.e., a contract is specified as a tuple $\langle A, G\rangle$, where A is the assumption and G is the guarantee that must be satisfied by the replacement. The idea is independent on the formalism which is used to represent A and G, the only requirement is that A and G have an equivalent trace semantic. The replacement I satisfies the contract if and only if $I \cap [A] \subseteq [G]$, where $[A]$ and $[G]$ represents the set of traces which corresponds to the assumption A and the guarantee G. The framework supports the contract decomposition when the system architecture is iteratively specified and the properties of interest are described with temporal logic. More precisely, the framework guarantees that if

[11] The LTS that describes the assumption is obtained from the DFA computed by the learning algorithm.

the contracts of the sub-components hold, then the contract of the parent component must also hold. Given a contract C for a component S, the set of the contracts $\mathcal{C}$ of the sub-components of S is a refinement of the contract C if and only if *a*) correct implementations of the sub-contracts $C_i \in \mathcal{C}$ are also correct implementation of the contract C; *b*) for every component i with sub-contract $C_i \in \mathcal{C}$ the correct implementations of the sub-contracts of the other components and a correct environment of S form a correct environment for i . In [30] the authors also discuss how to use their contract refinement framework in a compositional verification setting. In compositional verification the properties of the system are obtained by the properties of the sub-components without using the replacements (implementations) of the components. The procedure is based on the classical structure of deduction proof, i.e., starting from a set of axioms, the properties of the system are obtained iteratively applying a set of inference rules.

While in the classical synthesis problem the system reads all the signals generated by its environment, in [73] the authors consider the *synthesis with incomplete information* problem, which concerns the case in which each process can read only a part of the signals of the underling process. For example, in a distributed program the processes can read only the signals of the underlying processes. Given a set of readable signals I, a set of non readable signals E and the set of output signals O, the synthesis problem concerns the computation of a strategy $P : (2^I)^* \rightarrow 2^O$ where the computation of P are the infinite words over $2^{I \cup E \cup O}$.

In an uncertainty context, where non deterministic domains are considered, an execution may result in one or more sequences of states. Cimatti et al., [28] propose an algorithm that allows to compute (synthesize) plans. The goal is to compute plans which satisfy a reachability property, i.e., a condition on the final state of the execution of a plan. The planner guarantees that the plans provide a chance to reach the goal (“weak planning"), that the goal is always achieved (“strong planning") or the goal is achieved with trial-and-error strategies.

2.5 Checking the refinement

The goal of the refinement checking process is to verify after any change, i.e., when an incompleteness is refined, if the system possesses the properties of interest. Problems such as compositional reasoning, modular verification, component substitutability and hierarchical model checking are related to our work.

Compositional reasoning [70,89] tries to reduce the verification effort by decomposing the system into different components. The idea is to verify properties on individual components and, starting from these properties, infer the properties that hold in the whole system without the construction of the global state space. Given a system $\mathcal{M}$ composed by two components $\mathcal{M}_1$ and $\mathcal{M}_2$ executed in parallel, i.e., $\mathcal{M} = \mathcal{M}_1 \| \mathcal{M}_2$, and a property φ, compositional approaches try to independently analyze $\mathcal{M}_1$ and $\mathcal{M}_2$ to deduce that $\mathcal{M} \models \varphi$.

The most famous approach for compositional reasoning is the *assume-guarantee* paradigm [4, 69, 109]. In assume-guarantee, the classic Hoare style proof system [64] is extended by adding additional constraints on the context in which the process is executed, i.e., an assumption, and on the ways in which the program can modify the context in which it is performed, i.e., a guarantee. The assumption constraints the ways

in which the system can be changed by other programs, while the guarantee specifies the conditions the process is ensuring. In the assume guarantee paradigm, assumptions and guarantees hold during the whole process execution. Formally, $\langle true \rangle M \langle \phi \rangle$ and $\langle \phi \rangle M' \langle \psi \rangle$ implies that $\langle true \rangle M || M' \langle \psi \rangle$ [109]. The notation $\langle true \rangle M \langle \phi \rangle$ specifies that the process M guarantees the property ϕ in any environment since no assumptions are specified for M ($true$), while $\langle \phi \rangle M' \langle \psi \rangle$ states that the process M' guarantees the property ψ under the assumption that it is executed in an environment that satisfies ϕ. Informally, if the model M guarantees ϕ and the model M' guarantees ψ when it is located in an environment that satisfies ϕ, then when M and M' are executed in parallel, they satisfy ψ. This is proved without constructing the state space of $M||M'$.

Modular verification [124] is a particular instance of compositional reasoning which is focused on verification. It tries to exploit the natural decomposition of the system to check properties over the single components and infer the properties that hold in the global system. For example, a modular verification technique may *check* that $\langle \varphi \rangle M' \langle \psi \rangle$ and $\langle true \rangle M \langle \varphi \rangle$ holds, and starting from these two results infer that $\langle true \rangle M || M' \langle \psi \rangle$ using an assume guarantee reasoning style.

Grumberg and Long [59] proposed a verification method that supports modular verification. They develop a framework which is based on *a*) ∀CTL a subset of the Computation Tree Logic (CTL) for which satisfaction is preserved under composition. ∀CTL does not include the existential quantifiers and, to assure that the existential quantifiers are not generated via negation, assume that the formulae are expressed in negation normal form[12]; *b*) a pre-order relation $\preceq$ that captures the correspondence between the components and the system containing the components. The pre-order relation is based on the simulation relation[13] and supports an assume-guarantee reasoning style for the verification of the property. Given these two hypothesis, the model checking framework can be viewed as determining whether a formula is true in all the systems containing the component. For example, to verify whether $\mathcal{M}||\mathcal{M}' \models \phi$ it is possible to verify the following relations $\mathcal{M} \preceq \mathcal{A}$ and $\mathcal{A}||\mathcal{M}' \preceq \mathcal{A}'$ and $\mathcal{M}||\mathcal{A}' \models \phi$, i.e., $\mathcal{M}$ discharges the assumption $\mathcal{A}$, $\mathcal{A}||\mathcal{M}'$ discharges $\mathcal{A}'$ and $\mathcal{M}||\mathcal{A}'$ satisfies ϕ. The framework is exploited in the context of Moore machines and implemented in a symbolic framework.

Component substitutability [122] concerns the verification of a system when a component is removed from the running system (that can lead to unavailability of the previous provided services) and replaced by a new one (that can lead to violations of global correctness properties that were previously respected). The substitutability problem can then be associated with the verification of the following criteria: *a*) any updated portion of a software system must continue to provide the services offered by its earlier counterpart; *b*) previously established correctness properties must remain valid for the new version of the software system. The main idea behind component substitutability is to reduce the amount of time and effort required to verify an entire system after each (even minor) software update by localizing the verification effort to single components.

In [18, 121, 122] the component substitutability problem is solved by the use of two model checking techniques. Given a set of components $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots \mathcal{C}_n\}$ and a set of new components $\mathcal{C}' = \{\mathcal{C}'_i \mid i \in \mathcal{I}\}$ and $\mathcal{I} \subseteq \{1, \ldots n\}$ the problem is to check

[12] In negation normal form the negations are only applied to atomic propositions.

[13] Intuitively, a structure $\mathcal{M}$ can be simulated by a structure $\mathcal{M}'$ if every state s of $\mathcal{M}$ can be simulated by a state s' of $\mathcal{M}'$. A state s' simulates a state s if their labels agree on the atomic propositions and every run starting from s corresponds with a run starting from s'.

whether a safety property ϕ holds in the new assembly. The idea it to reduce the verification effort of subsequent verifications exploiting the results of the previous one by focusing only on the portions of the system (components) that have changed. First, the algorithm checks that the behaviors of $\mathcal{C}_i$ are a subset of the behaviors exhibited by $\mathcal{C}'_i$, i.e., it checks containment. Checking containment is usually an expensive activity in terms of time. For this reason the proposed procedure exploits an under and over approximation technique which is based on abstractions [17,57]. The idea it to create two models $\mathcal{M}_i$ and $\mathcal{M}'_i$ such that $\mathcal{C}_i \subseteq \mathcal{M}_i \subseteq \mathcal{M}'_i \subseteq \mathcal{C}'_i$, where $\subseteq$ means less behaviors. If $\mathcal{C}'_i$ is a valid substitute for $\mathcal{C}_i$ the procedure terminates otherwise a counterexample is returned. If the counterexample is valid the property is not satisfied, otherwise it is a spurious counterexample due to the approximations performed. At this point a dynamic regular set learning technique in conjunction with the assume-guarantee reasoning algorithm is used to refine $\mathcal{M}_i$. The overall approach has been implemented in the $COMFORT$ [19] reasoning framework. The experiments confirm the effectiveness of the dynamic approach with respect to the complete verification of the model of the system after any update.

Model checking of *Hierarchical State Machines* (HSMs) with respect to LTL properties has been considered in several works, such as [6–8]. In these works the authors analyze sequential HSMs[14] and propose a solution which avoids the exponential blow up necessary for computing the flattened version K^F of the HSM. Given a HSM K and an automata A which may be obtained from an LTL formula, the model checking problem is to solve the automata-emptiness problem, i.e., to check whether $\mathcal{L}(A) \cap \mathcal{L}(K^F)$ is empty, where K^F is the expanded version of K. If $\mathcal{L}(A) \cap \mathcal{L}(K^F)$ is not empty, every word in $\mathcal{L}(A) \cap \mathcal{L}(K^F)$ is a “bad" behavior. The automata-emptiness is solved by reduction to a *cycle* detection problem. The idea is to pair each HSM K_i against the automaton A for every possible way of pairing an entry node of K_i with a state of A. This implies that every exit node of K_i can be paired with every state of A, which causes an overhead of $|A|^2$ in the verification procedure. Then, an emptiness checking algorithm can be applied on this new structure. Thus, the temporal complexity of the verification algorithm is $\mathcal{O}(|K| \cdot |A|^3)$ (for ordinary FSMs it would be $\mathcal{O}(|K| \cdot |A|)$) where $|K|$ is the size of the HSM obtained by summing the cardinality of the set of the superstates, the normal states and the transitions, and $|A|$ is the size of the Büchi Automaton obtained from the LTL property. Thus, if we consider the LTL formula ϕ, the procedure has a temporal complexity $\mathcal{O}(|K| \cdot 8^{|\phi|})$, due to the overhead necessary to convert a LTL formula into the corresponding BA. Note that a HSM can be analyzed only when all the refinements levels have been specified, i.e., the analysis process must be conducted only at the end of the software development cycle. The authors also demonstrate that the model checking problem of CTL properties can be solved in time $\mathcal{O}(|K| \cdot 2^{|\phi| \cdot d})$, where d is the maximum number of exit nodes[15] of each HSMs. The verification of CTL properties requires to compute all the states that satisfy a particular sub-formula. The authors also prove that for a single-exit node FSM the problem of checking a CTL formula is PSPACE-complete in the size of the formula.

Famelis et al., [44] consider uncertainty as “multiple possibilities" (alternatives). Rather than having a single replacement associated with a component, the developer

[14] The authors only consider sequential FSM, that is they do not consider parallel execution.

[15] The exit nodes of a FSM $\mathcal{M}$ are the states that are connected to the states of a higher level of the hierarchical FSM.

has a set of possible replacements and he/she is not sure on which is the correct one. This implies an uncertainty also in the requirements the system must satisfy. To handle uncertainty the authors propose the use of annotations to precisely encode sets of possible models. These models are encoded into a propositional logic formula, where propositions encode the presence of states and transitions into the model. The properties are expressed into first order logic (FOL) and predicates over the structure of the graph. The reasoning mechanism is able to answer three different questions: does the property hold for all, some or none of the alternatives? If the property does not hold why is it so? If the property is a necessary constraint how to filter the alternatives for which it gets violated? The model checking algorithm returns true, false and maybe. The true value specifies that the property holds in all the concretizations of the model, false means that it does not hold for any of them, maybe is returned when the property holds only in some of the concretizations. This type of model checking is also indicated as *through* checking. The model checking algorithm exploits a SAT solver to check whether the propositional logic formulae obtained by combining the model of the system with the requirements and the negation of the requirements are satisfiable. If the property and its negation are satisfiable there exists a concretization where the property holds and one where it does not hold, thus the answer is maybe. Otherwise, if the requirement or its negation is satisfiable, the answer is true or false, respectively.

Salay et al., [118] proposed a methodology to verify if a partial model $\mathcal{M}'$ is a refinement of $\mathcal{M}$. The procedure first encodes $\mathcal{M}$ and $\mathcal{M}'$ into First order logic (FOL) formula and checks if the encoding of $\mathcal{M}'$ is satisfiable, that is the model is consistent and there exists a concretization, i.e., a valid refinement. Then, the algorithm checks that $\mathcal{M}'$ has no more concretizations than $\mathcal{M}$, i.e., it proves that the formula representing the encoding of $\mathcal{M}'$ implies the formula representing the encoding of $\mathcal{M}$. Furthermore, [118] also analyzes the problem of verifying a *refining transformation*. A refining transformation is a change in the model which allows the reduction of its uncertainty. Given a transformation (i.e., a refinement rule that can be iteratively applied), the objective is to analyze the result obtained by applying the rule repeatedly until it can no longer be applied. Salay et al., [118] realized a prototype tool which uses TXL [35] to translate the ecore models into first order encodings which are analyzed using Alloy [68].

Larsen and Steffen, [76] present a *constraint*-oriented proof (verification) methodology for MTSs. The procedure is based on the following steps: *a*) description of the system constraints, that is, the MTSs specifying particular behaviors of the system, i.e., projective views. Each projective view (constraint) specifies the behaviors of the system with respect to a specific parameter configuration (input value) through required transition. In each view, the behavior of the system with respect to the other values is described using -a possible infinite number of- admissible transitions; *b*) separation of the property to be verified in a conjunction of sub-properties which refer to the different projective views that can be independently verified (for this reason the procedure is said to be based on the separation of proof of obligations). The transformation generates a proof condition which is a First Order Logic formula in a Skolem normal form and depends on a single Skolem constant. *c*) verification of a single refinement (MTS) with respect to the Skolem constant. Note that, since the refinement may contain an infinite number of transitions, an abstraction technique can be used. The abstraction concerns

the identification of an equivalence relation to encode the infinite system into a finite one.

Santone et al., [119] given a formal specification of an incomplete system $\mathcal{M}$, try to characterize the "collaborators" of $\mathcal{M}$ that, given the communication interface $\mathcal{I}$, guarantee that the property ϕ is satisfied. The communication interface $\mathcal{I}$ is a sub-set of actions used by $\mathcal{M}$ to communicate with the collaborators. The idea is to find a formula (an *assumption*) ϕ', such that for every $\mathcal{M}'$ that satisfies ϕ', $(\mathcal{M} \| \mathcal{M}') \setminus \mathcal{I} \models \phi$. The desired property ($\phi$) and the synthesized one (ϕ') are expressed through Selective Hennessy-Milner logic (SHML) formula, which allows to specify branching temporal logic formula in the context of CCSs. To generate the *assumption*, the authors propose a *tableau based* approach. Tableau based approaches try to find a proof of the satisfaction of a property ϕ in $\mathcal{M}$ using a top-down reasoning mechanism. The authors extend the tableau based approach by separating the property ϕ (goal) and the environment, where at each step the goal specifies the part of the formula still to be proved, while the environment records the solution produced along the branch. At each step the tableau method simplifies the goal to be proved and extends the environment until a terminal sequence is reached.

CHAPTER 3

Background

"Ignorance is the curse of God; knowledge is the wing wherewith we fly to heaven."

William Shakespeare, 1564-1616

This chapter reviews the baselines on which this work is founded. It is a short introduction that contains the background necessary for reading the following chapters of this thesis[1]. Section 3.1 describes Finite State Automata (FSAs) and Büchi Automata (BAs), two of the most used modeling formalisms to describe the system under development. Section 3.2 presents Linear Time Temporal Logic (LTL) and its semantic. LTL is one of the modeling languages commonly used to specify the software properties of interest. Finally, Section 3.3 discusses how the automata based model checking framework allows proving that the system under development possesses the properties of interest.

3.1 Modeling systems

The modeling formalism used to describe the system under development strongly depends on the type of the system and on the properties the developer wants to describe. In this work we consider systems which are *sequentials*. A sequential system is characterized by a single execution thread and a state which changes in relation with the inputs of the system[2]. This section describes Finite State Automata (FSAs) and Büchi Automata (BAs), two of the most used modeling formalisms used to describe sequential systems.

[1] The interested reader may refer to [9, 33] for additional information.

[2] Differently from sequential systems, concurrent systems involve different components (usually called processes) which run in parallel.

3.1.1 Finite State Automata

Finite State Automata (FSAs) are a widely used modeling formalism which describes systems through a finite set of states and transitions among them. *States* represent snapshots of the system configurations. *Transitions* specify the possible evolutions of the system, describing how the state of the system changes over time. Transitions are labeled with *atomic propositions*, i.e., the statements that are true when the transitions are performed.

Definition 3.1.1 (Finite State Automaton [87,93])**.** *Given a finite set of atomic propositions AP, a* non-deterministic finite state automaton *(FSA) over finite words is a tuple $\langle \Sigma, Q, \Delta, Q^0, F \rangle$, where a) $\Sigma = 2^{AP}$ is the finite alphabet; b) Q is the finite set of states; c) $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation; d) $Q^0 \subseteq Q$ is the set of initial states; e) $F \subseteq Q$ is the set of final states.*

An automaton is non-deterministic whenever reading an input letter $\sigma \in \Sigma$, it is possible to reach two different states of the automaton or when the FSA has more than one initial state. An example of FSA defined over the set of atomic propositions $AP = \{start,\ send, wait, timeout, ack, fail, ok, abort,\ success\}$ is shown in Figure 3.1. The sets $Q = \{q_1, q_2, \ldots, q_{13}\}$, $Q^0 = \{q_1\}$ and $F = \{q_2, q_3\}$ define the set of states, initial states (graphically marked with an incoming arrow) and final states (graphically marked with a double circle) of the automaton, respectively[3]. Note that, since the alphabet of the FSA corresponds to $\Sigma = 2^{AP}$, a transition can be labeled with any subset of atomic propositions. For example, a transition labeled with the propositions $start$ and $send$ is performed if and only if both the propositions are satisfied.

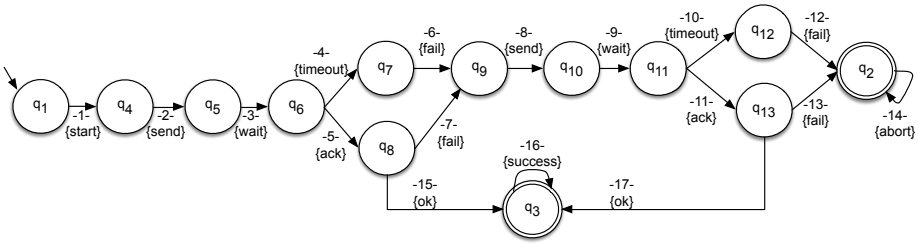


Figure 3.1: *A finite state automaton.*

Given a word $v \in \Sigma^*$ of length $|v|$, a run defines the sequence of states traversed by the automaton to recognize v. Formally,

Definition 3.1.2 (FSA run [33])**.** *Given a* word $v = v_0 v_1 v_2 \ldots v_{|v-1|}$ *of length $|v|$ in Σ^* and a FSA $\mathcal{M} = \langle \Sigma, Q, \Delta, Q^0, F \rangle$, a* run *over the word v is a mapping $\rho^* : \{0, 1, 2 \ldots |v|\} \rightarrow Q$ such that: a) $\rho^*(0) \in Q^0$; b) for all $0 \leq i < |v|$, $(\rho^*(i),\ v_i,\ \rho^*(i+1)) \in \Delta$.*

Informally, a run ρ^* corresponds to a path in the FSA $\mathcal{M}$, such that the first state $\rho^*(0)$ of the path is an initial state of $\mathcal{M}$, i.e., it is in the set Q^0, and the system moves from a state $\rho^*(i)$ to the next state of the path $\rho^*(i+1)$ by reading v_i. For example, the word $v = \{start\}.\{send\}.\{wait\}.\{timeout\}$ corresponds to the run ρ^* of the FSA

[3] In Figure 3.1 and in the rest of the thesis the transitions are also marked with a number which has no semantic value, but will be used only to refer to the transitions in the text.

$\mathcal{M}$ described in Figure 3.1 defined as follows: $\rho^*(0) = q_1$, $\rho^*(1) = q_4$, $\rho^*(2) = q_5$, $\rho^*(3) = q_6$, $\rho^*(4) = q_7$.

Definition 3.1.3 (Accepting run [33])**.** *A run ρ^* is* accepting *if and only if it ends in a final state of $\mathcal{M}$, i.e., $\rho^*(|v|) \in F$.*

Given a word v, the automaton $\mathcal{M}$ accepts v if and only if there exists an accepting run of $\mathcal{M}$ over v. For example, the finite word $v = \{start\}.\ \{send\}.\{wait\}.\{ack\}.\{ok\}$ is accepted by the automaton described in Figure 3.1 through a path that goes into the states q_1, q_4, q_5, q_6, q_8, q_3.

Definition 3.1.4 (Language of a FSA [33])**.** *The language $\mathcal{L}^*(\mathcal{M}) \subseteq \Sigma^*$ associated with the automaton $\mathcal{M}$ contains all the words that are accepted by $\mathcal{M}$.*

The size $|\mathcal{M}|$ of a FSA $\mathcal{M}$ is the sum of the cardinality of the set of its states and the set of its transitions. Formally,

Definition 3.1.5 (Size of a FSA)**.** *The size $|\mathcal{M}|$ of a FSA $\mathcal{M}$ is $|Q| + |\Delta|$.*

3.1.2 Büchi Automata

Software systems may exhibit both finite and infinite behaviors. In the former case the systems are designed to stop their execution whenever a final state is reached, in the latter their execution never terminates. The same theoretical framework can be applied in both the cases, i.e., by translating finite behaviors into infinite ones. This can be done by extending the alphabet of the FSA with a predefined character (e.g., no_op) that is not in the original alphabet and represents an operation that can be always executed in its final states whenever the other transitions are disabled[4]. Thus, it is only necessary to consider finite automata over infinite words. An infinite word, also called ω-*word* is a word that contains at least a sub-word that is repeated infinitely many times. The most famous class of automata over infinite words are Büchi automata (BAs).

Definition 3.1.6 (Büchi automaton [14])**.** *A* non-deterministic Büchi automaton *(BA) is a FSA $\langle \Sigma, Q, \Delta, Q^0, F \rangle$ where the set of final states F of the FSA is used to define the acceptance condition for infinite words (also called* ω-words*). Hence, for Büchi automata F is usually called the set of* accepting states*.*

Given an ω-*word* $v = v_0 v_1 v_2 \ldots$ a run defines an execution of the BA (sequence of states).

Definition 3.1.7 (BA run [33])**.** *Given an* ω-word $v = v_0 v_1 v_2 \ldots$ *in Σ^ω and a BA $\langle \Sigma, Q, \Delta, Q^0, F \rangle$, a* run *over v is a mapping $\rho^\omega : \{0, 1, 2 \ldots\} \rightarrow Q$ such that: a) $\rho^\omega(0) \in Q^0$; b) for all $i \geq 0$, $(\rho^\omega(i), v_i, \rho^\omega(i+1)) \in \Delta$.*

Note that, when Büchi automata are considered, the domain of ρ^ω refers to the whole set of natural numbers. We denote as $inf(\rho^\omega)$ the set of states that appear infinitely often in the run ρ^ω. For example, when the automaton $\mathcal{M}$ depicted in Figure 3.1 is interpreted as a BA, the word $\{start\}.\{send\}.\ \{wait\}.\{ack\}.\{ok\}.\{success\}^\omega$ is associated with the run ρ^ω, such that $\rho^\omega(0) = q_1$, $\rho^\omega(1) = q_4$, $\rho^\omega(2) = q_5$, $\rho^\omega(3) = q_6$, $\rho^\omega(4) = q_8$ and $\forall i > 4, \rho^\omega(i) = q_3$.

[4]This technique is also indicated in literature as stuttering [74, 102, 103, 105].

Definition 3.1.8 (Accepting run [33]). *A run ρ^ω of a BA $\mathcal{M}$ is* accepting *if and only if* $inf(\rho^\omega) \cap F \neq \emptyset$

Definition 3.1.8 requires that an accepting run contains at least one accepting state that appears in the run ρ^ω infinitely often. An *ω-word* v is accepted by a BA $\mathcal{M}$ iff there exists an accepting run associated with v. For example, if we consider the automaton depicted in Figure 3.1 as BA, the word $\{start\}.\{send\}.\{wait\}.\{ack\}.\{ok\}.\{success\}^\omega$ is accepted since the accepting state q_3 is entered an infinite number of times.

Definition 3.1.9 (Language of a BA [33]). *The language $\mathcal{L}^\omega(\mathcal{M}) \subseteq \Sigma^\omega$ of a BA $\mathcal{M}$ consists of all the* ω-words *accepted by $\mathcal{M}$.*

In general, when a BA is used to represent the system, all the states in Q are set as accepting. The size of a BA corresponds to the size of the corresponding FSA.

3.2 Modeling requirements

After the developer has designed a model of the system, he/she may want to check whether it possesses certain properties, i.e., it satisfies the requirements of interest. Natural languages are one of the most frequently used ways to express requirements. The main problem over the use of natural languages in the context of formal verification, is ambiguity. Ambiguity concerns the possibility of interpreting the requirements in different ways. Vice versa, to allow formal verification, requirements must be specified in a modeling formalism which is unambiguous and has a semantic which is compatible to the one used for describing the model, i.e., it must be possible to precisely identify the set of models that satisfy and do not satisfy the properties of interest.

3.2.1 Linear Time Temporal Logic

Linear Time Temporal Logic (LTL) [111] provides an intuitive and mathematically precise notation for expressing properties of the software execution. Linear Time Temporal Logic extends propositional logic with modalities that allow the specification of "temporal" relations about events. LTL formulae are obtained by combining atomic propositions with the boolean connectors $\wedge$ (and) and $\neg$ (not) and the temporal modalities X (next) and U (until).

Definition 3.2.1 (LTL syntax). *Given a set AP of atomic propositions (with $a \in AP$), a LTL formula ϕ is formed according to the following grammar:*

$$\phi = true \mid a \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid X\phi \mid \phi_1 U \phi_2$$

where ϕ_1 and ϕ_2 are LTL formulae.

The or ($\vee$), implication ($\rightarrow$) and equivalence ($\leftrightarrow$) boolean connectors can be derived using the operators and ($\wedge$) and not ($\neg$). Furthermore, the temporal operators eventually (F) and globally (G), which specify that the property must hold sometimes and always in the future, respectively, can be expressed in terms of the until operator as $(true)U(\phi)$ and $\neg(F(\neg\phi))$[5]. For example, the LTL formula $G(send \rightarrow F(success))$ defined over the propositions $send$ and $success$, specifies that always whenever a message is sent

[5]The interested reader may refer to [9] for additional information.

(the proposition $send$ is true), the message is finally delivered (the proposition $success$ is true).

Definition 3.2.2 (LTL semantic over words)**.** *Given the alphabet $\Sigma = 2^{AP}$ and an ω-word $v = v_0 v_1 v_2 \ldots$ in Σ^ω, the satisfaction relation $\models$ is defined by:*

$$
\begin{array}{lll}
v & \models true & \\
v & \models a & \Leftrightarrow a \in v_0 \\
v & \models \phi_1 \wedge \phi_2 & \Leftrightarrow v \models \phi_1 \textit{ and } v \models \phi_2 \\
v & \models \neg\phi & \Leftrightarrow v \not\models \phi \\
v & \models X\phi & \Leftrightarrow v^1 \models \phi \\
v & \models \phi_1 U \phi_2 & \Leftrightarrow \exists j \geq 0 \mid (v^j \models \phi_2 \textit{ and } \forall i \mid 0 \leq i < j,\ v^i \models \phi_1)
\end{array}
$$

where $v^i = v_i v_{i+1} \ldots$ is the suffix of v starting from the character i.

An ω-word v satisfies the LTL formula ϕ when $v \models \phi$. The ω-language $\mathcal{L}^\omega(\phi)$ defined by ϕ is the set of all possible words that satisfy ϕ, i.e., $\mathcal{L}^\omega(\phi) = \{v \mid v \models \phi\}$. For example, the language associated with the formula $G(send \rightarrow F(success))$ contains all the words such that every $send$ proposition is followed by a $success$.

3.2.2 Büchi Automata

Instead of using LTL, the developer can also specify the properties (claims) of interest using directly Büchi automata [33]. One of the advantages of this choice is that both the model of the system and its claims are represented in the same formalism [105]. Furthermore, in several cases, it is easier to specify the properties of interest as an automaton rather than an LTL formula. In these cases, the developer can provide an automaton $\mathcal{A}_\phi$ which contains the set of all the allowed behaviors, or the automaton $\overline{\mathcal{A}_\phi}$ which contains the set of violating (undesirable) behaviors. The language $\mathcal{L}^\omega(\mathcal{A}_\phi)$ ($\mathcal{L}^\omega(\overline{\mathcal{A}_\phi})$) of the automaton $\mathcal{A}_\phi$ ($\overline{\mathcal{A}_\phi}$) contains all the allowed (disallowed) behaviors. Note that, when claims are considered, the edges of the BA are labeled with boolean expressions rather than subsets of atomic propositions AP. For example, in a BA defined over the alphabet $\Sigma = \{a, b\}$, an edge labeled with the boolean expression $a \vee (\neg b)$ represents the set of transitions labeled with $\{a\}$, $\{a, b\}$ and $\{\emptyset\}$.

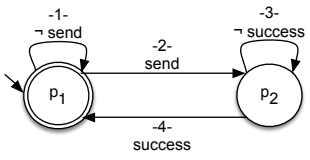


Figure 3.2: *The Büchi automaton corresponding to the property $G(send \rightarrow F(success))$.*

Note that, each LTL formula can be converted into an equivalent BA which specifies the claim of interest. For example, the property $G(send \rightarrow F(success))$ can be converted into the automaton represented in Figure 3.2. The language of the automaton contains all the words that satisfy the LTL formula, i.e., all the words in which a sending request is followed by a success.

3.3 Automata based Model checking

This section recalls the main steps of the classical automata based model checking procedure [138] which checks the model of the system against the corresponding claim. The procedure assumes the model of the system and the claim specified through a BA and an LTL formula, respectively.

Given a BA $\mathcal{M}$ and a LTL property ϕ, the model $\mathcal{M}$ satisfies ϕ if and only if $\mathcal{L}^{\omega}(\mathcal{M}) \subseteq \mathcal{L}^{\omega}(\phi)$. The condition $\mathcal{L}^{\omega}(\mathcal{M}) \subseteq \mathcal{L}^{\omega}(\phi)$ states that each behavior of the model of the system must be contained in the set of the behaviors allowed by the property. As mentioned in Section 3.2.2 each LTL formula ϕ can be converted into a BA $\mathcal{A}_{\phi}$ which recognizes the same language. Thus, a BA $\mathcal{M}$ satisfies an LTL property ϕ if and only if $\mathcal{L}^{\omega}(\mathcal{M}) \subseteq \mathcal{L}^{\omega}(\mathcal{A}_{\phi})$. Equivalently, $\mathcal{M}$ satisfies ϕ if and only if $\mathcal{L}^{\omega}(\mathcal{M}) \cap \mathcal{L}^{\omega}(\overline{\mathcal{A}_{\phi}}) = \emptyset$, where $\mathcal{L}^{\omega}(\overline{\mathcal{A}_{\phi}})$ specifies the set of behaviors not allowed by ϕ, i.e., the complement of the automaton $\mathcal{A}_{\phi}$. The formula $\mathcal{L}^{\omega}(\mathcal{M}) \cap \mathcal{L}^{\omega}(\overline{\mathcal{A}_{\phi}}) = \emptyset$ specifies that there are no behaviors of $\mathcal{M}$ disallowed by ϕ. This observation allows using the algorithm to check the emptiness of the language obtained from the intersection of two Büchi automata for the verification of the property satisfaction. Checking the emptiness of an automaton is simpler than checking the language containment. Furthermore, when ϕ is expressed as an LTL formula we can avoid the complementation of $\mathcal{A}_{\phi}$ for obtaining $\overline{\mathcal{A}_{\phi}}$, since $\overline{\mathcal{A}_{\phi}}$ can be directly obtained by translating $\neg\phi$ into the corresponding automaton. This is more efficient than translating ϕ and then computing the complement of the corresponding automaton.

The automata-based model checking procedure is based on the previous observations and on the fact that BAs are closed under intersection and complement [14]. The procedure is defined over three steps:

- translating the negation of the formula ϕ into an equivalent BA $\overline{\mathcal{A}_{\phi}}$;
- computing the intersection automaton $\mathcal{I}$ between the BA $\overline{\mathcal{A}_{\phi}}$ and the model $\mathcal{M}$;
- check the emptiness of the intersection automaton $\mathcal{I}$.

1) transforming the negation of the LTL formula ϕ into the corresponding automaton. The negation of a LTL formula ϕ can be converted into an equivalent BA $\overline{\mathcal{A}_{\phi}}$, where the automaton $\overline{\mathcal{A}_{\phi}}$ encodes the set of behaviors forbidden by the property. This automaton can be designed manually[6], or can be constructed starting from the LTL formula $\neg\phi$, where ϕ is the property to be verified. In the latter case, the procedure can be executed with a time complexity $\mathcal{O}(2^{(|\neg\phi|)})$, where $|\neg\phi|$ is the size of the formula $\neg\phi$ [52]. There are different algorithms to convert LTL formulae into the corresponding BAs (e.g., [53]), which are not described here since they are out from the scope of this thesis. Figure 3.3 describes the BA corresponding to the negation of the property

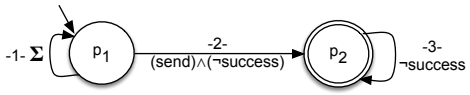


Figure 3.3: *The BA $\overline{\mathcal{A}_{\phi}}$ corresponding to the LTL property $\neg G(send \rightarrow F(success))$.*

[6]This assumption was also made by the earliest versions of the SPIN model checking tool [65].

$G(send \rightarrow F(success))$. The automaton recognizes all the words that contain a $send$ followed by an infinite number of characters which do not correspond to the symbol $success$[7].
2) computing the intersection automaton. Given the model $\mathcal{M}$, whose language $\mathcal{L}(\mathcal{M})$ contains all the possible behaviors exhibited by the system, and the automaton $\overline{\mathcal{A}_\phi}$, whose language $\mathcal{L}(\overline{\mathcal{A}_\phi})$ contains all the words that violate ϕ, the automaton $\mathcal{I}$ contains the set of behaviors of the model forbidden by the property, i.e., $\mathcal{L}(\mathcal{I}) = \mathcal{L}(\mathcal{M}) \cap \mathcal{L}(\overline{\mathcal{A}_\phi})$.

Definition 3.3.1 (Intersection automaton)**.** *Let $\mathcal{M} = \langle \Sigma, Q_\mathcal{M}, \Delta_\mathcal{M}, Q^0_\mathcal{M}, F_\mathcal{M} \rangle$ and $\overline{\mathcal{A}_\phi} = \langle \Sigma, Q_{\overline{\mathcal{A}_\phi}}, \Delta_{\overline{\mathcal{A}_\phi}}, Q^0_{\overline{\mathcal{A}_\phi}}, F_{\overline{\mathcal{A}_\phi}} \rangle$ be two BAs defined over the same alphabet Σ, the intersection automaton $\mathcal{I} = \mathcal{M} \cap \overline{\mathcal{A}_\phi}$ is the automaton $\langle \Sigma_\mathcal{I}, Q_\mathcal{I}, \Delta_\mathcal{I}, Q^0_\mathcal{I}, F_\mathcal{I} \rangle$, such that:*

- $\Sigma_\mathcal{I} = \Sigma$ *is the alphabet of the intersection automaton;*
- $Q_\mathcal{I} = Q_\mathcal{M} \times Q_{\overline{\mathcal{A}_\phi}} \times \{0, 1, 2\}$ *is the set of the states of the intersection automaton;*
- $\Delta_\mathcal{I}$ *is the transition relation such that* $(\langle q_i, p_j, x \rangle, a, \langle q_m, p_n, y \rangle) \in \Delta_\mathcal{I}$ *if and only if* $(q_i, a, q_m) \in \Delta_\mathcal{M}$ *and* $(p_j, a, p_n) \in \Delta_{\overline{\mathcal{A}_\phi}}$*. Moreover, each transition in* $\Delta_\mathcal{I}$ *must satisfy the following conditions:*
  - *if* $x = 0$ *and* $q_m \in F_\mathcal{M}$*, then* $y = 1$.
  - *if* $x = 1$ *and* $p_n \in F_{\overline{\mathcal{A}_\phi}}$*, then* $y = 2$.
  - *if* $x = 2$ *then* $y = 0$.
  - *otherwise,* $y = x$*;*
- $Q^0_\mathcal{I} = Q^0_\mathcal{M} \times Q^0_{\overline{\mathcal{A}_\phi}} \times \{0\}$ *is the set of initial states;*
- $F_\mathcal{I} = F_\mathcal{M} \times F_{\overline{\mathcal{A}_\phi}} \times \{2\}$ *is the set of accepting states.*

The alphabet of the intersection automaton corresponds to the alphabet of $\mathcal{M}$ and $\overline{\mathcal{A}_\phi}$. Each state of the intersection automaton is obtained by combining a state of $\mathcal{M}$ and $\overline{\mathcal{A}_\phi}$. The labels 0, 1 and 2 indicate that no accepting state is entered, at least one accepting state of $\mathcal{M}$ is entered and at least one accepting state of $\mathcal{M}$ and one of $\overline{\mathcal{A}_\phi}$ are entered, respectively. A transition $(\langle q_i, p_j, x \rangle, a, \langle q_m, p_n, y \rangle)$ is in the set $\Delta_\mathcal{I}$, if the components agree on the transitions to be performed. Note that, the transition relation guarantees that a run of the intersection automaton is accepting if an only if an accepting state of $\mathcal{M}$ and an accepting state of $\overline{\mathcal{A}_\phi}$ are entered infinitely often. For this reason the label of the states of the intersection automaton is initially 0. It changes from 0 to 1 and from 1 to 2 whenever an accepting state of the model and of the automaton that corresponds to the negation of the claim is entered, respectively. After an accepting state of $\mathcal{I}$ is visited it is set back to 0. When all the states of the model are considered as accepting, the set of accepting states of the intersection automaton simply contains all the states obtained by combining a state of the model with an accepting state of $\overline{\mathcal{A}_\phi}$.

[7]The Σ character is used to identify all the characters of the alphabet.

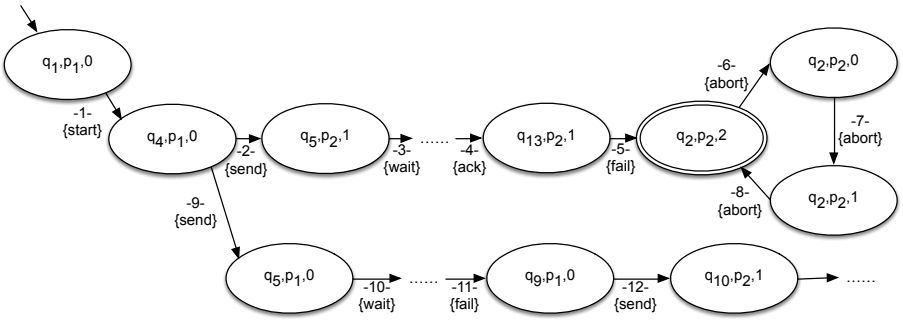


Figure 3.4: *A portion of the intersection automaton obtained from the BAs specified in Figure 3.1 and 3.3, respectively.*

The intersection procedure builds an intersection automaton that contains (in the worst case) $3 \cdot |Q_{\mathcal{M}}| \cdot |Q_{\overline{\mathcal{A}_\phi}}|$ states [25, 26], where $|Q_{\mathcal{M}}|$ and $|Q_{\overline{\mathcal{A}_\phi}}|$ are the number of states of $\mathcal{M}$ and $\overline{\mathcal{A}_\phi}$, respectively[8];

Figure 3.4 presents a portion of the state space of the intersection automaton obtained from the BAs specified in Figures 3.1 and 3.3, respectively. As previously mentioned, the transitions of the intersection automaton are obtained by the synchronous execution of the transitions of the model and the claim. For example, the transition 1 of the intersection automaton is generated by firing the transitions 1 of the model and the claim, respectively, while the transition 9 is generated by the synchronization of the transitions 2 and 1 of the two automata.

3) check the emptiness of the intersection automaton $\mathcal{I}$. by checking the emptiness of $\mathcal{I}$ it is possible to verify whether the property is satisfied or not in the model. If $\mathcal{I}$ is empty, the property is satisfied, otherwise every infinite word in the intersection automaton is a counterexample. The emptiness problem is usually solved through a double depth first search (DFS) algorithm [36, 128], with a linear time complexity $\mathcal{O}(|Q_{\mathcal{I}}| + |\Delta_{\mathcal{I}}|)$ [43]. $|Q_{\mathcal{I}}|$ and $|\Delta_{\mathcal{I}}|$ are the number of states and transitions of the intersection automaton $\mathcal{I}$. The double depth first search is based on a simple observation: if there is an (infinite) accepting run ρ^ω in a BA $\mathcal{I}$, then, there is a suffix $\rho^{\omega'}$ of ρ^ω that appears infinitely many times in ρ^ω . This suffix is associated with a strongly connected component (SCC) of the graph representing the automaton $\mathcal{I}$, which contains at least an accepting state. The first DFS identifies the accepting states of $\mathcal{I}$ ($s \in F_{\mathcal{I}}$) that are reachable from the initial states; the second DFS explores the graph searching for loops that involve these accepting states. If the second DFS detects a loop, a counterexample is returned, since an accepting run has been detected.

For example, the intersection automaton described in Figure 3.4 is not empty. Indeed, there are several words, such as $\{start\}.\{send\}.\{wait\}.\{timeout\}.\{fail\}.\{send\}.\{wait\}.\{timeout\}.\{fail\}.\{abort\}^\omega$, included in $\mathcal{L}^\omega(\mathcal{M})$ and $\mathcal{L}^\omega(\overline{\mathcal{A}_\phi})$.

If the property ϕ is expressed as an LTL formula, the *time complexity* of the model checking procedure is exponential in the length of the formula and linear in the size of the model[9], while *space complexity* is polynomial in the size of the specification and

[8] Other approaches presented in literature can create more succinct automata where, in the worst case, the number of the states is $2 \cdot |Q_{\mathcal{M}}| \cdot |Q_{\overline{\mathcal{A}_\phi}}|$ (see for example [100]).

[9] Time complexity is considered to be acceptable since the properties expressed in LTL are usually concise [138].

poly-logarithmic ($\mathcal{O}(log^2 n)$) in the size of the model [138] [82]. This is one of the main advantages of the automata-based model checking framework: it separates the hardest part of the problem, creating the automaton $\overline{\mathcal{A}_\phi}$ from the LTL formula ϕ, from the easier part, building the intersection automaton and solving the emptiness problem [138].

CHAPTER 4

Modeling Incomplete and Evolving Systems

"If I were again beginning my studies, I would follow the advice of Plato and start with mathematics."

Galileo Galilei, 1564-1642

Software development is an iterative process which includes a set of development steps that transform the initial high level specification of the system into its final, fully specified, implementation [142]. The modeling formalism used in this refinement process depends on the properties of the system that are of interest. This chapter proposes two modeling formalisms that support incompleteness when the functional behavior of a sequential system is considered. Note that in a sequential system the computation starts from one of its initial states. Then, the state of the system changes by firing transitions which make the system moving from one state to another.

Section 4.1 describes Incomplete Finite State Automata (IFSAs) and Incomplete Büchi Automata (IBAs), two modeling formalisms introduced in this work to support incomplete specifications. Section 4.2 describes how these two modeling formalisms can be used in the refinement process, i.e., how the initial, incomplete, high level design of the system can be iteratively refined. Finally, Section 4.3 specifies how the LTL formula which describes the property over the functional behavior of the system can be interpreted over IBAs.

4.1 Modeling incomplete systems

This section proposes Incomplete FSAs (IFSAs) and Incomplete BAs (IBAs) two modeling formalisms which extend Finite State Automata (FSAs) and Büchi Automata (BAs) to support incompleteness.

4.1.1 Incomplete Finite State Automata

Incomplete FSAs (IFSAs) are a state based modeling formalism that extends FSAs by partitioning the set of the states Q in two sets: the set of *regular* states R and the set of *black box states* states B[1]. Regular states correspond to classical automata states, while black box states are placeholders for configurations in which the behavior of the system is currently unspecified and will be later refined by other automata, i.e., other IFSAs. In the rest of the thesis black box states are often abbreviated as black boxes or boxes.

Definition 4.1.1 (Incomplete Finite State Automaton)**.** *Given a finite set of atomic propositions AP, a non-deterministic Incomplete Finite State Automaton (IFSA) $\mathcal{M}$ is a tuple $\langle \Sigma, R, B, Q, \Delta, Q^0, F \rangle$, where: a) $\Sigma = 2^{AP}$ is the finite alphabet; b) R is the finite set of* regular *states; c) B is the finite set of* box *states; d) Q is the finite set of states such that $Q = B \cup R$ and $B \cap R = \emptyset$; e) $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation; f) $Q^0 \subseteq Q$ is the set of initial states; g) $F \subseteq Q$ is the set of final states.*

Graphically, boxes are filled with black, initial states are marked by an incoming arrow, and final states are double circled. Note that the transition relation allows the definition of transitions that connect states of Q irrespective of their type. An example of IFSA defined over the set of propositions $AP = \{start,\ fail, ok, success, abort\}$ is shown in Figure 4.1. This automaton is a well known example of incompleteness in the context of software development and has been presented in [8]. $Q = \{q_1,\ send_1, send_2, q_2,\ q_3\}$, $Q^0 = \{q_1\}$, $F = \{q_2, q_3\}$ and $B = \{send_1, send_2\}$ are the set of the states, of the initial states, of the final states and of the boxes, respectively.

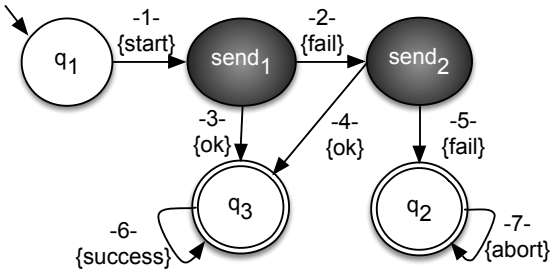


Figure 4.1: *An example of IFSA.*

Given a word $v \in \Sigma^*$ of length $|v|$ a run defines the sequences of states traversed by the automaton to recognize v.

Definition 4.1.2 (IFSA run)**.** *Given a set of atomic propositions AP, an IFSA $\mathcal{M} = \langle \Sigma, R, B, Q, \Delta, Q^0, F \rangle$, such that $\Sigma = 2^{AP}$, a set of atomic propositions AP', such that $AP \subseteq AP'$ and $\Sigma' = 2^{AP'}$, and a* word $v = v_0 v_1 v_2 \dots v_{|v-1|}$ *of length $|v|$ in Σ'^*, a* run *over the word v is a mapping $\rho^* : \{0, 1, 2 \dots |v|\} \rightarrow Q$ such that: a) $\rho^*(0) \in Q^0$; b) for all $0 \leq i < |v|$, $(\rho^*(i), v_i, \rho^*(i+1)) \in \Delta$ or $\rho^*(i) \in B$ and $\rho^*(i) = \rho^*(i+1)$.*

A run ρ^* corresponds to a path in the IFSA $\mathcal{M}$, such that the first state $\rho^*(0)$ of the path is an initial state of $\mathcal{M}$, i.e., it is in the set Q^0, and either the system moves

[1] Black box states have been also identified in other works as transparent states, such as in [120].

form a state $\rho^*(i)$ to the next state $\rho^*(i+1)$ by reading the character v_i, or the state $\rho^*(i)$ is a box ($\rho^*(i) \in B$) and the character v_i is recognized "inside" the box $\rho^*(i) = \rho^*(i+1)$. For example, the finite word $\{start\}.\{send\}.\{fail\}$ can be associated with the run $\rho^*(0) = q_1$, $\rho^*(1) = send_1$, $\rho^*(2) = send_1$ and $\rho^*(3) = send_2$ or with the run $\rho^*(0) = q_1$, $\rho^*(1) = send_1$, $\rho^*(2) = send_1$ and $\rho^*(3) = send_1$.

Definition 4.1.3 (IFSA definitely accepting and possibly accepting run)**.** *A run ρ^* is* definitely accepting *if and only if $\rho^*(|v|) \in F$ and for all $0 \leq i \leq |v|$, $\rho^*(i) \in R$. A run ρ^* is* possibly accepting *if and only if $\rho^*(|v|) \in F$ and there exists $0 \leq i \leq |v|$ such that $\rho^*(i) \in B$. A run ρ^* is not accepting otherwise.*

Informally, a run ρ^* is *definitely accepting* if and only if ends in a final state of $\mathcal{M}$ and all the states of the run are regular, it is *possibly accepting* if and only if it ends in a final state of $\mathcal{M}$ and there exists at least a state of the run which is a box, it is not accepting otherwise.

Definition 4.1.4 (IFSA definitely accepted and possibly accepted word)**.** *An IFSA $\mathcal{M}$* definitely accepts *a word v if and only if there exists a definitely accepting run of $\mathcal{M}$ on v. $\mathcal{M}$* possibly accepts *a word v if and only if it does not definitely accept v and there exists at least a possibly accepting run of $\mathcal{M}$ on v. Finally, $\mathcal{M}$* does not accept *v iff it does not contain any definitely accepting or possibly accepting run for v.*

Note that possibly accepted words describe *possible behaviors*. For example, the word $\{start\}.\{send\}.\{ok\}$ is possibly accepted by the automaton presented in Figure 4.1 since no definitely accepting run exists, while it exists a possibly accepting run described by the function ρ^*, such that $\rho^*(0) = q_1$, $\rho^*(1) = send_1$, $\rho^*(2) = send_1$ and $\rho^*(3) = q_3$

Definition 4.1.5 (IFSA definitely accepted and possibly accepted language)**.** *Given a finite set of atomic propositions AP', such that $AP \subseteq AP'$, and the alphabet $\Sigma' = 2^{AP'}$, the language $\mathcal{L}^*(\mathcal{M}) \subseteq \Sigma'^*$ definitely accepted by an IFSA $\mathcal{M}$ contains all the words $v_1, v_2 \dots v_n \in \Sigma'^*$ definitely accepted by $\mathcal{M}$. The possibly accepted language $\mathcal{L}_p^*(\mathcal{M}) \subseteq \Sigma^*$ of $\mathcal{M}$ contains all the words $v_1, v_2 \dots v_n \in \Sigma'^*$ possibly accepted by $\mathcal{M}$.*

Given an IFSA $\mathcal{M}$ it is possible to define its completion $\mathcal{M}_c$ as the FSA obtained by removing its boxes and their incoming and outgoing transitions.

Definition 4.1.6 (Completion of an IFSA)**.** *Given an IFSA $\mathcal{M} = \langle \Sigma, R, B, Q, \Delta, Q^0, F \rangle$ the completion of $\mathcal{M}$ is the FSA $\mathcal{M}_c = \langle \Sigma, R, \Delta_c, Q^0 \cap R, F \cap R \rangle$, such as $\Delta_c = \{(s, a, s') \mid (s, a, s') \in \Delta$ and $s \in R$ and $s' \in R\}$.*

It is possible to prove that the completion of an IFSA recognizes its definitely accepted language.

Lemma 4.1.1 (Language of the completion of an IFSA)**.** *Given an IFSA $\mathcal{M} = \langle \Sigma, R, B, Q, \Delta, Q^0, F \rangle$ the completion $\mathcal{M}_c$ of $\mathcal{M}$ recognizes the definitely accepted language $\mathcal{L}^*(\mathcal{M})$.*

Proof. To proof Lemma 4.1.1 it is necessary to demonstrate that a word is recognized by the completion if and only if it belongs to the definitely accepted language of $\mathcal{M}$, i.e., $v \in \mathcal{L}^*(\mathcal{M}) \Leftrightarrow v \in \mathcal{L}^*(\mathcal{M}_c)$.

($\Rightarrow$) Each word v accepted by $\mathcal{M}$ is associated with an accepting run ρ^* which contains only regular states. Since $\mathcal{M}_c$ contains all the regular states of $\mathcal{M}$ and the same transitions between these states, it is possible to simulate the run ρ^* of $\mathcal{M}$ on the automaton $\mathcal{M}_c$. This implies that v is definitely accepted by $\mathcal{M}_c$.

($\Leftarrow$) is proved by contradiction. Imagine that there exists a word v in $\mathcal{L}^*(\mathcal{M}_c)$ which is not in $\mathcal{L}^*(\mathcal{M})$. This implies that there exists a run ρ^* in $\mathcal{M}_c$ which does not correspond to a run $\rho^{*'}$ in $\mathcal{M}$. Given one of the states $\rho^*(i)$ it can be associated to the corresponding state of $\mathcal{M}$. Given two states $\rho^*(i)$ and $\rho^*(i+1)$ of the run and the transition $(\rho^*(i), a, \rho^*(i+1)) \in \Delta_c$, it is possible to "simulate" the transition by performing the corresponding transition of $\mathcal{M}$ since $\Delta_c \subseteq \Delta$. This implies that v is also accepted by $\mathcal{M}$, and therefore it is in the language $\mathcal{L}^*(\mathcal{M})$, which violates the hypothesis. □

The size $|\mathcal{M}|$ of an IFSA $\mathcal{M}$ is the sum of the cardinality of the set of its states and the set of its transitions.

Definition 4.1.7 (Size of an IFSA)**.** *The size $|\mathcal{M}|$ of an IFSA $\mathcal{M}$ is $|Q| + |\Delta|$.*

4.1.2 Incomplete Büchi Automata

As specified in Section 3.1.2 software systems are usually not designed to stop during their execution, thus infinite models of computation are usually considered. This section introduces Incomplete BAs (IBAs) an extended version of BAs that support incompleteness.

Definition 4.1.8 (Incomplete BA)**.** *A non-deterministic incomplete Büchi automaton (IBA) is an IFSA $\langle \Sigma, R, B, Q, \Delta, Q^0, F \rangle$, where the set of final states F of the IFSA is used to define the acceptance condition for infinite words (also called ω-*words*). As in the case of BAs, F identifies the set of* accepting states*.*

Given an *ω-word* $v = v_0 v_1 v_2 \ldots$ a run defines an execution of the IBA (sequence of states).

Definition 4.1.9 (IBA run)**.** *Given a set of atomic propositions AP, an IFSA $\mathcal{M} = \langle \Sigma, R, B, Q, \Delta, Q^0, F \rangle$, such that $\Sigma = 2^{AP}$, a set of atomic propositions AP', such that $AP \subseteq AP'$ and $\Sigma' = 2^{AP'}$, and a word $v \in \Sigma'^{\omega}$, a* run *$\rho^\omega : \{0, 1, 2, \ldots\} \to Q$ over v is defined for an IBA as follows: a) $\rho^\omega(0) \in Q^0$; b) for all $i \geq 0$, $(\rho^\omega(i), v_i, \rho^\omega(i+1)) \in \Delta$ or $\rho^\omega(i) \in B$ and $\rho^\omega(i) = \rho^\omega(i+1)$.*

Informally, a character v_i of the word v can be recognized by a transition of the IBA, changing the state of the automaton from $\rho^\omega(i)$ to $\rho^\omega(i+1)$, or it can be recognized by a transition of the IBA that will replace the box $\rho^\omega(i) \in B$. In the latter, the state $\rho^\omega(i+1)$ of the automaton after the recognition of v_i, corresponds to $\rho^\omega(i)$, since the automaton remains inside the automaton which corresponds with the box $\rho^\omega(i)$. For example, the infinite word $\{start\}.\{send\}.\{ok\}.\{success\}^\omega$ can be associated with the run $\rho^\omega(0) = q_1$, $\rho^\omega(1) = send_1$ and $\rho^\omega(2) = send_1$ or and $\forall i \geq 3, \rho^\omega(i) = q_3$ of the automaton described in Figure 4.1 when it is interpreted as an IBA.

Let $inf(\rho^\omega)$ be the set of states that appear infinitely often in the run ρ^ω.

Definition 4.1.10 (IBA definitely accepted and possibly accepted run)**.** *A run ρ^ω of an IBA $\mathcal{M}$ is:* a) definitely accepting *if and only if $inf(\rho^\omega) \cap F \neq \emptyset$ and for all*

$i \geq 0, \rho^\omega(i) \in R$; *b*) possibly accepting *if and only if* $(inf(\rho^\omega) \cap F \neq \emptyset)$ *and exists* $\exists i \geq 0 \mid \rho^\omega(i) \in B$; *c*) not accepting *otherwise.*

Informally, a run is definitely accepting if some accepting states appear in ρ^ω infinitely often and all states of the run are regular states, it is possibly accepting if some accepting states appear in ρ^ω infinitely often and there is at least one state in the run that is a box, not accepting otherwise.

Definition 4.1.11 (IBA definitely accepted and possibly accepted word)**.** *An automaton* $\mathcal{M}$ definitely accepts *a word* v *if and only if there exists an accepting run of* $\mathcal{M}$ *on* v. $\mathcal{M}$ possibly accepts *a word* v *if and only if it does not definitely accept* v *and there exists at least a possibly accepting run of* $\mathcal{M}$ *on* v. *Finally,* $\mathcal{M}$ does not accept v *iff it does not contain any accepting or possibly accepting run for* v.

As for IFSA, possibly accepted words describe *possible behaviors*. For example, the automaton described in Figure 4.1 (when it is interpreted as an IBA) possibly accepts the infinite word $\{start\}.\{send\}.\{ok\}.\{success\}^\omega$ since a definitely accepting run does not exist but there exists a run which is possibly accepting.

Definition 4.1.12 (IBA definitely accepted and possibly accepted language)**.** *Given a finite set of atomic propositions* AP', *such that* $AP \subseteq AP'$, *and the alphabet* $\Sigma' = 2^{AP'}$, *the language* $\mathcal{L}^\omega(\mathcal{M}) \in \Sigma'^\omega$ *definitely accepted by an IBA* $\mathcal{M}$ *contains all the words definitely accepted by* $\mathcal{M}$. *The possibly accepted language* $\mathcal{L}_p^\omega(\mathcal{M}) \in \Sigma'^\omega$ *of* $\mathcal{M}$ *contains all the words possibly accepted by* $\mathcal{M}$.

The language $\mathcal{L}^\omega(\mathcal{M})$ can be defined by considering the BA $\mathcal{M}_c$ obtained from $\mathcal{M}$ by removing its boxes and their incoming and outgoing transitions.

Definition 4.1.13 (Completion of an IBA)**.** *Given an IBA* $\mathcal{M} = \langle \Sigma, R, B, Q, \Delta, Q^0, F \rangle$ *the completion of* $\mathcal{M}$ *is the BA* $\mathcal{M}_c = \langle \Sigma, R, \Delta_c, Q^0 \cap R, F \cap R \rangle$, *such as* $\Delta_c = \{(s, a, s') \mid (s, a, s') \in \Delta$ *and* $s \in R$ *and* $s' \in R\}$.

As for IFSA, it is possible to prove that the completion of an IBA recognizes its definitely accepted language.

Lemma 4.1.2 (Language of the completion of an IBA)**.** *Given an IBA* $\mathcal{M} = \langle \Sigma, R, B, Q, \Delta, Q^0, F \rangle$ *the completion* $\mathcal{M}_c$ *of* $\mathcal{M}$ *recognizes the definitely accepted language* $\mathcal{L}^\omega(\mathcal{M})$.

Proof. The prove of Lemma 4.1.2 is similar to the proof of Lemma 4.1.1 and requires to demonstrate that $v \in \mathcal{L}^\omega(\mathcal{M}) \Leftrightarrow v \in \mathcal{L}^\omega(\mathcal{M}_c)$.

($\Rightarrow$) Each word v definitely accepted by $\mathcal{M}$ is associated to a definitely accepting run ρ^ω which only contains regular states. Since $\mathcal{M}_c$ contains all the regular states of $\mathcal{M}$ and the same transitions between these states, it is possible to simulate the run ρ^ω of $\mathcal{M}$ on the automaton $\mathcal{M}_c$ which implies that v is accepted.

($\Leftarrow$) is proved by contradiction. Imagine that there exists a word v in $\mathcal{L}^\omega(\mathcal{M}_c)$ which is not in $\mathcal{L}^\omega(\mathcal{M})$. This implies that there exists a run ρ^ω in $\mathcal{M}_c$ which does not correspond to a run $\rho^{\omega'}$ in $\mathcal{M}$. Consider the run ρ^ω, each state $\rho^\omega(i)$ can be associated to the corresponding state of $\mathcal{M}$. Given two states $\rho^\omega(i)$ and $\rho^\omega(i+1)$ of the run and the transition $(\rho^\omega(i), a, \rho^\omega(i+1)) \in \Delta_c$ it is possible to "simulate" the transition by performing the corresponding transition of $\mathcal{M}$ since $\Delta_c \subseteq \Delta$. This implies that v is

also accepted by $\mathcal{M}$, and therefore it is in the language $\mathcal{L}^{\omega}(\mathcal{M})$, which violates the hypothesis. □

The size $|\mathcal{M}|$ of an IBA $\mathcal{M}$ is the sum of the cardinality of the set of its states and the set of its transitions.

Definition 4.1.14 (Size of an IBA)**.** *The size $|\mathcal{M}|$ of an IBA $\mathcal{M}$ is $|Q| + |\Delta|$.*

4.2 Refining incomplete models

The development activity is an iterative and incremental process through which the initial, high level, design $\mathcal{M}$ is iteratively refined. After having provided the initial high level model $\mathcal{M}$, the development activity proceeds through a set of refinement rounds $\mathcal{RR}$. At reach refinement round $i \in \mathcal{RR}$ a box b of $\mathcal{M}$ is refined. We use the term *refinement* to capture the notion of model elaboration, i.e., the model $\mathcal{N}$ is a refinement $\mathcal{M}$ if it is obtained from $\mathcal{M}$ by adding knowledge about the behavior of the system inside one of its boxes. We call *replacement* the sub-automaton which specifies the behavior of the system inside a specific box.

4.2.1 Refining Incomplete Büchi Automata

The refinement relation $\preceq$ allows the iterative concretization of the model of the system by replacing boxes with other IBAs. These IBAs are called *replacements*. The definition of the refinement relation $\preceq$ has been inspired from [123].

Definition 4.2.1 (Refinement)**.** *Let $\wp_{\mathcal{M}}$ the set of all possible IBAs. An (I)BA $\mathcal{N}$ is a refinement of an IBA $\mathcal{M}$, i.e., $\mathcal{M} \preceq \mathcal{N}$, iff $\Sigma_{\mathcal{M}} \subseteq \Sigma_{\mathcal{N}}$ and there exists some refinement relation $\Re \in Q_{\mathcal{M}} \times Q_{\mathcal{N}}$, such that:*

1. *for all $q_{\mathcal{M}} \in R_{\mathcal{M}}$ there exists exactly one $q_{\mathcal{N}} \in R_{\mathcal{N}}$ such that $(q^0_{\mathcal{M}}, q^0_{\mathcal{N}}) \in \Re$;*
2. *for all $q_{\mathcal{N}} \in Q_{\mathcal{N}}$ there exists exactly one $q_{\mathcal{M}} \in Q_{\mathcal{M}}$ such that $(q_{\mathcal{M}}, q_{\mathcal{N}}) \in \Re$;*
3. *for all $(q_{\mathcal{M}}, q_{\mathcal{N}}) \in \Re$, if $q_{\mathcal{N}} \in Q^0_{\mathcal{N}}$ then $q_{\mathcal{M}} \in Q^0_{\mathcal{M}}$;*
4. *for all $(q_{\mathcal{M}}, q_{\mathcal{N}}) \in \Re$, if $q_{\mathcal{N}} \in B_{\mathcal{N}}$ then $q_{\mathcal{M}} \in B_{\mathcal{M}}$;*
5. *for all $(q_{\mathcal{M}}, q_{\mathcal{N}}) \in \Re$, if $q_{\mathcal{N}} \in F_{\mathcal{N}}$ then $q_{\mathcal{M}} \in F_{\mathcal{M}}$;*
6. *for all $(q_{\mathcal{M}}, q_{\mathcal{N}}) \in \Re$, if $q_{\mathcal{M}} \in Q^0_{\mathcal{M}} \cap R_{\mathcal{M}}$ then $q_{\mathcal{N}} \in Q^0_{\mathcal{N}} \cap R_{\mathcal{N}}$;*
7. *for all $(q_{\mathcal{M}}, q_{\mathcal{N}}) \in \Re$, if $q_{\mathcal{M}} \in F_{\mathcal{M}} \cap R_{\mathcal{M}}$ then $q_{\mathcal{N}} \in F_{\mathcal{N}}$;*
8. *for all $(q_{\mathcal{M}}, q_{\mathcal{N}}) \in \Re$ and $\forall a \in \Sigma_{\mathcal{N}}$, if $(q_{\mathcal{M}}, a, q'_{\mathcal{M}}) \in \Delta_{\mathcal{M}}$ then there exists $q'_{\mathcal{N}} \in Q_{\mathcal{N}}$ such that one of the following is satisfied:*
   - *$(q_{\mathcal{N}}, a, q'_{\mathcal{N}}) \in \Delta_{\mathcal{N}}$ and $(q'_{\mathcal{M}}, q'_{\mathcal{N}}) \in \Re$;*
   - *$q_{\mathcal{M}} \in B_{\mathcal{M}}$ and there exists $q''_{\mathcal{N}} \in Q_{\mathcal{N}}$ such that $(q_{\mathcal{M}}, q''_{\mathcal{N}}) \in \Re$ and $(q''_{\mathcal{N}}, a, q'_{\mathcal{N}}) \in \Delta_{\mathcal{N}}$;*
9. *for all $(q_{\mathcal{M}}, q_{\mathcal{N}}) \in \Re$ and $\forall a \in \Sigma_{\mathcal{N}}$, if $(q_{\mathcal{N}}, a, q'_{\mathcal{N}}) \in \Delta_{\mathcal{N}}$ one of the following holds:*

- *there exists* $q'_{\mathcal{M}} \in Q_{\mathcal{M}}$ *such that* $(q'_{\mathcal{M}}, q'_{\mathcal{N}}) \in \Re$ *and* $(q_{\mathcal{M}}, a, q'_{\mathcal{M}}) \in \Delta_{\mathcal{M}}$*;*
- $q_{\mathcal{M}} \in B_{\mathcal{M}}$ *and* $(q_{\mathcal{M}}, q'_{\mathcal{N}}) \in \Re$*.*

The idea behind the refinement relation is that every behavior of $\mathcal{M}$ *must be preserved* in its refinement $\mathcal{N}$, and every behavior of $\mathcal{N}$ must correspond to a behavior of $\mathcal{M}$.

Condition 1 imposes that each *regular state* of $\mathcal{M}$ is associated with exactly one regular state of the refinement $\mathcal{N}$. When $q_{\mathcal{M}}$ is a box several states (or none) of $\mathcal{N}$ can be associated with $q_{\mathcal{M}}$. Condition 2 imposes that each state (regular or black box) of the refinement $\mathcal{N}$ is associated with exactly one state of the model $\mathcal{M}$. Condition 3 specifies that any initial state of the refinement $\mathcal{N}$ is associated with an initial state of the model $\mathcal{M}$. Condition 4 guarantees that any *box* in the refinement $\mathcal{N}$ is associated with a box of the model $\mathcal{M}$, i.e., it is not possible to refine a regular state into a box. Condition 5 specifies that each *accepting state* of $\mathcal{N}$ corresponds with an accepting state of $\mathcal{M}$. Condition 6 forces each initial and regular state of the model $\mathcal{M}$ to be associated with an initial and regular state $\mathcal{N}$. Condition 7 specifies that each accepting and regular state of $\mathcal{M}$ is associated with an accepting and regular state of $\mathcal{N}$. Finally, conditions 8 and 9 constrain the *transition relation*. Given a state $q_{\mathcal{M}}$ in $\mathcal{M}$ and a corresponding state $q_{\mathcal{N}}$ of the refined automaton $\mathcal{N}$, conditions 8 specifies that for each transition $(q_{\mathcal{M}}, a, q'_{\mathcal{M}})$ either there exists a state $q'_{\mathcal{N}}$ that follows $q_{\mathcal{N}}$ through a transition labeled with a, or the state $q_{\mathcal{M}}$ is a box and another transition $(q''_{\mathcal{N}}, a, q'_{\mathcal{N}})$ that exits the state $q''_{\mathcal{N}}$ of the replacement of the box $q_{\mathcal{M}}$ is associated with the transition $(q_{\mathcal{M}}, a, q'_{\mathcal{M}})$[2]. Condition 9 guarantees that each transition $(q_{\mathcal{N}}, a, q'_{\mathcal{N}})$ in the refinement $\mathcal{N}$ must be associated with a transition $(q_{\mathcal{M}}, a, q'_{\mathcal{M}})$ of $\mathcal{M}$ or it is a transition of the replacement of the box $q_{\mathcal{M}}$, i.e., $q_{\mathcal{M}} \in B_{\mathcal{M}}$.

Consider for example the automaton $\mathcal{M}$ presented in Figure 4.1 and the automaton $\mathcal{N}$ of Figure 3.1, $\mathcal{M} \preceq \mathcal{N}$, through the relation $\Re = \{(q_1, q_1),\ (send_1, q_4), (send_1, q_5),\ (send_1, q_6),\ (send_1, q_7),\ (send_1, q_8),\ (send_2, q_9),\ (send_2, q_{10}),\ (send_2, q_{11}),\ (send_2, q_{12}), (send_2, q_{13}), (q_2, q_2),\ (q_3, q_3)\}$.

Definition 4.2.2 (Implementation)**.** *A BA* $\mathcal{N}$ *is an implementation of an IBA* $\mathcal{M}$ *if and only if* $\mathcal{M} \preceq \mathcal{N}$*.*

The automaton $\mathcal{N}$ presented in Figure 3.1 is also an implementation of the automaton $\mathcal{M}$ described in Figure 4.1.

It is important to notice that the refinement relation preserves the language containment relation, i.e., a possibly accepted word of $\mathcal{M}$ can be definitely accepted, possibly accepted or not accepted in the refinement, but every definitely accepted and not accepted word remains accepted or not accepted in $\mathcal{N}$.

Theorem 4.2.1 (Language preservation)**.** *Given a model* $\mathcal{M}$ *and one of its refinements* $\mathcal{N}$*, for all* $v^\omega \in \Sigma^\omega$*:*

1. *if* $v^\omega \in \mathcal{L}^\omega(\mathcal{M})$ *then* $v^\omega \in \mathcal{L}^\omega(\mathcal{N})$
2. *if* $v^\omega \notin (\mathcal{L}_p^\omega(\mathcal{M}) \cup \mathcal{L}^\omega(\mathcal{M}))$ *then* $v^\omega \notin (\mathcal{L}_p^\omega(\mathcal{N}) \cup \mathcal{L}^\omega(\mathcal{N}))$

[2]Note that the state $q''_{\mathcal{N}}$ must not be necessarily reachable in the replacement of the state $q_{\mathcal{N}}$.

Proof. Let us first prove the statement 1 of Theorem 1. Since v^ω is accepted by the IBA $\mathcal{L}^\omega(\mathcal{M})$, it must exists an accepting run $\rho^\omega_\mathcal{M}$ of $\mathcal{M}$. Note that accepting runs only contains states that are regular. Let us consider the initial state $\rho^\omega_\mathcal{M}(0)$. By Definition 4.2.1 conditions 1 and 6 it must exists a state $q^0_\mathcal{N} \in Q^0_\mathcal{N}$ such that $(q^0_\mathcal{M}, q^0_\mathcal{N}) \in \Re$. Let us identify with $\rho^\omega_\mathcal{N}$ a run which starts in this state and is iteratively obtained as follows. Consider a generic step step i. Given two states $\rho^\omega_\mathcal{M}(i)$, $\rho^\omega_\mathcal{M}(i+1)$ of the run $\rho^\omega_\mathcal{M}$ it must exist a transition $(\rho^\omega_\mathcal{M}(i), a, \rho^\omega_\mathcal{M}(i+1)) \in \Delta_\mathcal{M}$. By Definition 4.2.1 condition 8 it must exists a transition $(q^\omega_\mathcal{N}(i), a, q^\omega_\mathcal{N}(i+1))$ of $\Delta_\mathcal{N}$, where $(\rho^\omega_\mathcal{M}(i), \rho^\omega_\mathcal{N}(i)) \in \Re$ and $(\rho^\omega_\mathcal{M}(i+1), \rho^\omega_\mathcal{N}(i+1)) \in \Re$. Condition 7 imposes that a regular accepting state of $q_\mathcal{M}$ is associated with an accepting state of $q_\mathcal{N}$. Thus, since $\rho^\omega_\mathcal{M}$ and $\rho^\omega_\mathcal{N}$ move from $\rho^\omega_\mathcal{M}(i)$ and $\rho^\omega_\mathcal{N}(i)$ to $\rho^\omega_\mathcal{M}(i+1)$ and $\rho^\omega_\mathcal{N}(i+1)$ by reading the same characters and for construction the corresponding runs are accepting we conclude that $v^\omega \in \mathcal{L}^\omega(\mathcal{N})$.

Let us now consider the statement 2 of Theorem 1. The proof is by contradiction. Imagine that there exists a word $v^\omega \notin (\mathcal{L}^\omega_p(\mathcal{M}) \cup \mathcal{L}^\omega(\mathcal{M}))$ and $v^\omega \in (\mathcal{L}^\omega_p(\mathcal{N}) \cup \mathcal{L}^\omega(\mathcal{N}))$. Since $v^\omega \in (\mathcal{L}^\omega_p(\mathcal{N}) \cup \mathcal{L}^\omega(\mathcal{N}))$, it must exists an accepting or possibly accepting run $\rho^\omega_\mathcal{N}$ associated with this word. Let us consider the initial state $\rho^\omega_\mathcal{N}(0)$ of this run. By Definition 4.2.1, condition 2, it must exists a state $q_\mathcal{M} \in Q_\mathcal{M}$ such that $(q_\mathcal{M}, \rho^\omega_\mathcal{N}(0)) \in \Re$. Since $\rho_\mathcal{N}(0)$ is initial by Definition 4.2.1, condition 3, we derive that $q_\mathcal{M}$ is also initial. Let us identify as $\rho^\omega_\mathcal{M}$ a run in $\mathcal{M}$ which starts from $q_\mathcal{M}$. Given two states $\rho^\omega_\mathcal{N}(i)$, $\rho^\omega_\mathcal{N}(i+1)$ of the run $\rho^\omega_\mathcal{N}$ it must exists a transition $(\rho^\omega_\mathcal{N}(i), a, \rho^\omega_\mathcal{N}(i+1)) \in \Delta_\mathcal{N}$. By Definition 4.2.1, condition 9, either it exists a transition $(\rho^\omega_\mathcal{M}(i), a, \rho^\omega_\mathcal{M}(i+1))$ of $\Delta_\mathcal{M}$ or $\rho^\omega_\mathcal{M}(i) \in B_\mathcal{M}$. Finally, condition 5 imposes that an accepting state of $q_\mathcal{N}$ is associated with an accepting state of $q_\mathcal{M}$. Thus, since $\rho^\omega_\mathcal{M}$ and $\rho^\omega_\mathcal{N}$ moves from $\rho^\omega_\mathcal{M}(i)$ and $\rho^\omega_\mathcal{N}(i)$ to $\rho^\omega_\mathcal{M}(i+1)$ and $\rho^\omega_\mathcal{N}(i+1)$, respectively, by reading the same characters, or $\rho^\omega_\mathcal{M}(i) = \rho^\omega_\mathcal{M}(i+1)$ and $\rho^\omega_\mathcal{M}(i) \in B_\mathcal{M}$, and for construction the corresponding runs are accepting, we conclude that $v^\omega \in \mathcal{L}^\omega(\mathcal{M})$ or $v^\omega \in \mathcal{L}^\omega_p(\mathcal{M})$. □

4.2.2 Replacements

Consider an IBA $\mathcal{M}$. At each refinement round $i \in \mathcal{RR}$, the developer designs a replacement $\mathcal{R}_i$[3] for one of the boxes $b \in B_{\mathcal{M}_i}$ of $\mathcal{M}_i$, where $\mathcal{M}_i$ is the refinement of the automaton $\mathcal{M}$ before the refinement round i.

Definition 4.2.3 (Replacement)**.** *Given an IBA $\mathcal{M} = \langle \Sigma_\mathcal{M}, R_\mathcal{M}, B_\mathcal{M}, Q_\mathcal{M}, \Delta_\mathcal{M}, Q^0_\mathcal{M}, F_\mathcal{M} \rangle$, the replacement $\mathcal{R}_b$ of the box $b \in B_\mathcal{M}$ is defined as a triple $\langle \mathcal{M}_b, \Delta^{inR_b}, \Delta^{outR_b} \rangle$. $\mathcal{M}_b = \langle \Sigma_{\mathcal{M}_b}, R_{\mathcal{M}_b}, B_{\mathcal{M}_b}, Q_{\mathcal{M}_b}, \Delta_{\mathcal{M}_b}, Q^0_{\mathcal{M}_b}, F_{\mathcal{M}_b} \rangle$ is an (I)BA, $\Delta^{inR_b} \subseteq \{(q', a, q) \mid (q', a, b) \in \Delta_\mathcal{M}$ and $q \in Q_{\mathcal{M}_b}\}$ and $\Delta^{outR_b} \subseteq \{(q, a, q') \mid (b, a, q') \in \Delta_\mathcal{M}$ and $q \in Q_{\mathcal{M}_b}\}$ are its incoming and outgoing transitions, respectively. $\mathcal{R}_b$ must satisfy the following conditions:*

- *if $b \notin Q^0_\mathcal{M}$ then $Q^0_{\mathcal{M}_b} = \emptyset$;*
- *if $b \notin F_\mathcal{M}$ then $F_{\mathcal{M}_b} = \emptyset$;*
- *if $(q', a, b) \in \Delta_\mathcal{M}$ then it exists $(q', a, q) \in \Delta^{inR_b}$, such that $q \in Q_{\mathcal{M}_b}$;*
- *if $(b, a, q') \in \Delta_\mathcal{M}$ then it exists $(q, a, q') \in \Delta^{outR_b}$, such that $q \in Q_{\mathcal{M}_b}$;*
- *if $(b, a, b) \in \Delta_\mathcal{M}$ then it exists $(q', a, q) \in \Delta_{\mathcal{M}_b}$.*

[3]The term replacement is also used for example in [95].

Informally, $\mathcal{M}_b$ is the (I)BA to be substituted to the box b, Δ^{inR_b} and Δ^{outR_b} specify how the replacement is connected to the states of $\mathcal{M}$. Consider for example the replacement $\mathcal{R}_{send_1}$ described in Figure 4.2 which refers to the box $send_1$ of the model $\mathcal{M}$ described in Figure 4.1 (the replacement assumes that $send_1$ is bot initial and accepting). The automaton $\mathcal{M}_{send_1}$ is defined over the set of atomic propositions $AP_{send_1} = \{start,\ booting, ready, send, wait, timeout, ack, fail, ok\}$. The state q_{14}, q_{15} and q_{17} is the initial, accepting and a box of the replacement, respectively. Note that the initial/accepting states must be initial/accepting for the whole system, i.e., not only in the scope of the considered replacement. Furthermore, the destination/source of an incoming/outgoing transition is not considered as initial/accepting if they are not initial/accepting for $\mathcal{M}_{send_1}$.

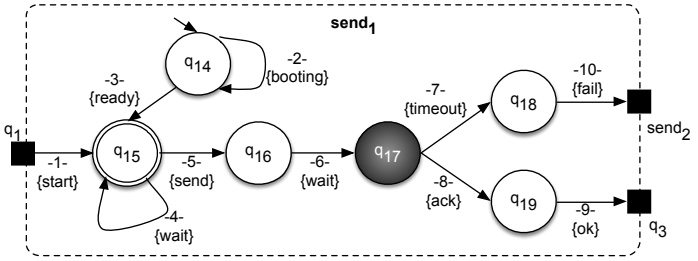


Figure 4.2: *The replacement of the box* $send_1$.

When a replacement is considered four different types of runs can be identified:

- *finite internal runs*: are the runs which start from an initial state that is internal to the replacement and reach an outgoing transition of the replacement;
- *infinite internal runs*: are the runs that start from an initial state that is internal to the replacement and infinitely enter an internal accepting state *without* leaving the replacement;
- *finite external runs*: are the runs that start from an incoming transition of the replacement and reach an outgoing transition of the replacement, i.e., they are finite paths that cross the component;
- *infinite external runs*: are the runs that start from an incoming transition of the replacement and reach an accepting state which is internal to the replacement itself *without* leaving the replacement.

We identify as $Q^{0in_b} = \{q \in Q_{\mathcal{M}}$ such that there exist $q' \in Q_{\mathcal{M}_b}$ and an $a \in \Sigma_{\mathcal{M}}$ and $(q, a, q') \in \Delta^{inR_b}\}$ and $Q^{0out_b} = \{q \in Q_{\mathcal{M}_b}$ such that there exist $q' \in Q_{\mathcal{M}_b}$ and an $a \in \Sigma_{\mathcal{M}}$ and $(q', a, q) \in \Delta^{inR_b}\}$ the set of the states that are sources and destinations of incoming transitions, respectively. We indicate with $F^{in_b} = \{q \in Q_{\mathcal{M}_b}$ such that there exist $q' \in Q_{\mathcal{M}}$ and an $a \in \Sigma_{\mathcal{M}}$ and $(q, a, q') \in \Delta^{outR_b}\}$ and with $F^{out_b} = \{q \in Q_{\mathcal{M}}$ such that there exist $q' \in Q_{\mathcal{M}_b}$ and an $a \in \Sigma_{\mathcal{M}}$ and $(q', a, q) \in \Delta^{outR_b}\}$ the set of the states that are sources and destinations of outgoing transitions.

Infinite internal runs, finite internal runs, infinite external runs and finite external runs can then formally defined as in the following.

Definition 4.2.4 (Finite Internal Run)**.** *Given a replacement* $\mathcal{R}_b = \langle \mathcal{M}_b, \Delta^{inR_b}, \Delta^{outR_b} \rangle$ *defined over the automaton* $\mathcal{M}_b = \langle \Sigma_{\mathcal{M}_b}, R_{\mathcal{M}_b}, B_{\mathcal{M}_b}, Q_{\mathcal{M}_b}, \Delta_{\mathcal{M}_b}, Q^0_{\mathcal{M}_b}, F_{\mathcal{M}_b} \rangle$ *a finite internal run* ρ_b^{f*} *over a word* $v \in \Sigma^*$ *is a finite run of the finite state automaton* $\mathcal{M}'_b = \langle \Sigma_{\mathcal{M}_b}, R_{\mathcal{M}_b}, B_{\mathcal{M}_b}, Q_{\mathcal{M}_b} \cup F^{out_b}, \Delta_{\mathcal{M}_b} \cup \Delta^{outR_b}, Q^0_s, F^{out_b} \rangle$.

A finite internal run is associated to the IFSA corresponding to the replacement where the initial states include only the internal initial states of the replacement and the final states are the destinations of its outgoing transitions. For example, the run $\rho^{f*}_{send_1}(\{ready\}.\{send\}.\{wait\}.\{timeout\}.\ \{fail\})$ where $\rho^{f*}_{send_1}(0) = q_{14}$, $\rho^{f*}_{send_1}(1) = q_{15}$, $\rho^{f*}_{send_1}(2) = q_{16}$, $\rho^{f*}_{send_1}(3) = q_{17}$, $\rho^{f*}_{send_1}(4) = q_{18}$, $\rho^{f*}_{send_1}(5) = send_2$, is a finite internal run of the replacement presented in Figure 4.2.

Definition 4.2.5 (Infinite Internal Run)**.** *Given a replacement* $\mathcal{R}_b = \langle \mathcal{M}_b, \Delta^{inR_b}, \Delta^{outR_b} \rangle$ *defined over the automaton* $\mathcal{M}_b = \langle \Sigma_{\mathcal{M}_b}, R_{\mathcal{M}_b}, B_{\mathcal{M}_b}, Q_{\mathcal{M}_b}, \Delta_{\mathcal{M}_b}, Q^0_{\mathcal{M}_b}, F_b \rangle$ *a infinite internal run* $\rho_b^{i\omega}$ *over a word* $v \in \Sigma^\omega$ *is an infinite run of the (Incomplete) Büchi automaton* $\mathcal{M}'_b = \langle \Sigma_{\mathcal{M}_b}, R_{\mathcal{M}_b}, B_{\mathcal{M}_b}, Q_{\mathcal{M}_b}, \Delta_{\mathcal{M}_b}, Q^0_{\mathcal{M}_b}, F_{\mathcal{M}_b} \rangle$.

An infinite internal run refers to the IBA obtained from the automaton $\mathcal{M}_b$ where the initial and accepting states include only the initial and accepting states of the automaton associated with the replacement. For example, the infinite internal run $\rho^{i\omega}_{send_1}(\{ready\}.\{wait\}^\omega)$ is a function such that $\rho^{i\omega}_{send_1}(0) = q_{14}$, and $\forall i > 1, \rho^{i\omega}_{send_1}(i) = q_{15}$.

Definition 4.2.6 (Finite External Run)**.** *Given a replacement* $\mathcal{R}_b = \langle \mathcal{M}_b, \Delta^{inR_b}, \Delta^{outR_b} \rangle$ *defined over the automaton* $\mathcal{M}_b = \langle \Sigma_{\mathcal{M}_b}, R_{\mathcal{M}_b}, B_{\mathcal{M}_b}, Q_{\mathcal{M}_b}, \Delta_{\mathcal{M}_b}, Q^0_{\mathcal{M}_b}, F_{\mathcal{M}_b} \rangle$ *a finite external run* ρ_b^{e*} *over a word* $v \in \Sigma^*$ *is a finite run of the finite state automaton* $\mathcal{M}'_b = \langle \Sigma_{\mathcal{M}_b}, R_{\mathcal{M}_b}, B_{\mathcal{M}_b}, Q_{\mathcal{M}_b} \cup Q^{0in_b} \cup F^{out_b}, \Delta_{\mathcal{M}_b} \cup \Delta^{inR_b} \cup \Delta^{outR_b}, Q^{0in_b}, F^{out_b} \rangle$.

A finite external run refers to the IFSA obtained from the automaton $\mathcal{M}_b$ where the initial and accepting states include only the sources and the destinations of the incoming and outgoing transitions, respectively. For example, the finite external run $\rho^{e*}_{send_1}(\{start\}.\{send\}.\{wait\}.\{timeout\}.\ \{fail\})$ is a function such that $\rho^{e*}_{send_1}(0) = q_1$, $\rho^{e*}_{send_1}(1) = q_{15}$, $\rho^{e*}_{send_1}(2) = q_{16}$, $\rho^{e*}_{send_1}(3) = q_{17}$, $\rho^{e*}_{send_1}(4) = q_{18}$ and $\rho^{e*}_{send_1}(5) = send_2$.

Definition 4.2.7 (Infinite External Run)**.** *Given a replacement* $\mathcal{R}_b = \langle \mathcal{M}_b, \Delta^{inR_b}, \Delta^{outR_b} \rangle$ *defined over the automaton* $\mathcal{M}_b = \langle \Sigma_{\mathcal{M}_b}, R_{\mathcal{M}_b}, B_{\mathcal{M}_b}, Q_{\mathcal{M}_b}, \Delta_{\mathcal{M}_b}, Q^0_{\mathcal{M}_b}, F_{\mathcal{M}_b} \rangle$ *a infinite external run* $\rho_b^{e\omega}$ *over a word* $v \in \Sigma^\omega$ *is an infinite run of the (Incomplete) Büchi automaton* $\mathcal{M}'_b = \langle \Sigma_{\mathcal{M}_b}, R_{\mathcal{M}_b}, B_{\mathcal{M}_b}, Q_{\mathcal{M}_b} \cup Q^{0in_b}, \Delta_{\mathcal{M}_b} \cup \Delta^{inR_b}, Q^{0in_b}, F_{\mathcal{M}_b} \rangle$.

An infinite external run refers to the IBA obtained from the automaton $\mathcal{M}_b$ where the initial states include the source states of the incoming transitions and the accepting states contains only the accepting states of $\mathcal{M}_b$. For example, the infinite external run $\rho^{e\omega}_{send_1}(\{start\}.\{wait\}^\omega)$ is a function such that $\rho^{e\omega}_{send_1}(0) = q_1$ and $\forall i \geq 1, \rho^{e\omega}_{send_1}(i) = q_{15}$.

Given the four types of runs previously described, which are defined over IFSA and IBA, it is possible to distinguish between the three types of finite/infinite runs described in Sections 4.1.1 and 4.1.2: *definitely accepting, possibly accepting and not accepting*. For example, the replacement presented in Figure 4.2 contains two types of *definitely accepting infinite* runs. The infinite *internal* runs involve the states q_{14} and

q_{15}, i.e., they recognize all the words in the form $\{booting\}^*.\{ready\}.\{wait\}^\omega$. The infinite *external* runs involve the states q_1 and q_{15} and recognize all the words in the form $\{start\}.\{wait\}^\omega$. Furthermore, the replacement contains two types of *possibly accepting finite* runs. The finite *internal* possibly accepting runs includes all the runs which involve the states q_{14}, q_{15}, q_{16}, q_{17} and q_{18} or q_{19}, respectively. The finite *external* possibly accepting runs includes all the runs which involve the states q_1, q_{15}, q_{16}, q_{17} and q_{18} or q_{19}, respectively.

Let us now discuss the language recognized by a replacement. The replacement $\mathcal{R}_b$ *internally definitely accepts* the finite word $v \in \Sigma^*$ if and only if there exists an internal finite definitely accepting run of $\mathcal{R}_b$ on v. The language of the finite words internally definitely accepted by the replacement $\mathcal{R}_b$ is indicated as $\mathcal{L}^{i*}(\mathcal{R}_b)$. The replacement $\mathcal{R}_b$ *externally definitely accepts* the finite word $v \in \Sigma^*$ if and only if there exists an external finite definitely accepting run of $\mathcal{R}_b$ on v. The language of the finite words externally definitely accepted by the replacement $\mathcal{R}_b$ is indicated as $\mathcal{L}^{e*}(\mathcal{R}_b)$. The replacement $\mathcal{R}_b$ *internally definitely accepts* the infinite word $v \in \Sigma^\omega$ if and only if there exists an internal infinite definitely accepting run of $\mathcal{R}_b$ on v. The language of the infinite words internally definitely accepted by the replacement $\mathcal{R}_b$ is indicated as $\mathcal{L}^{i\omega}(\mathcal{R}_b)$. The replacement $\mathcal{R}_b$ *externally definitely accepts* the infinite word $v \in \Sigma^\omega$ if and only if there exists an external infinite definitely accepting run of $\mathcal{R}_b$ on v. The language of the infinite words externally definitely accepted by the replacement $\mathcal{R}_b$ is indicated as $\mathcal{L}^{e\omega}(\mathcal{R}_b)$.

Let us now consider *possibly accepting words*. The replacement $\mathcal{R}_b$ *internally possibly accepts* the finite word $v \in \Sigma^*$ if and only if there exists an internal possibly finite accepting run of $\mathcal{R}_b$ on v. The language of the finite words internally possibly accepted by the replacement $\mathcal{R}_b$ is indicated as $\mathcal{L}_p^{i*}(\mathcal{R}_b)$. The replacement $\mathcal{R}_b$ *externally possibly accepts* the finite word $v \in \Sigma^*$ if and only if there exists an external finite possibly accepting run of $\mathcal{R}_b$ on v. The language of the finite words externally possibly accepted by the replacement $\mathcal{R}_b$ is indicated as $\mathcal{L}_p^{e*}(\mathcal{R}_b)$. The replacement $\mathcal{R}_b$ *internally possibly accepts* the infinite word $v \in \Sigma^\omega$ if and only if there exists an internal possibly infinite accepting run of $\mathcal{R}_b$ on v. The language of the infinite words internally possibly accepted by the replacement $\mathcal{R}_b$ is indicated as $\mathcal{L}_p^{i\omega}(\mathcal{R}_b)$. The replacement $\mathcal{R}_b$ *externally possibly accepts* the infinite word $v \in \Sigma^\omega$ if and only if there exists an external infinite possibly accepting run of $\mathcal{R}_b$ on v. The language of the infinite words externally possibly accepted by the replacement $\mathcal{R}_b$ is indicated as $\mathcal{L}_p^{e\omega}(\mathcal{R}_b)$.

As for IBA we define the *completion* of a replacement $\mathcal{R}_{b_c}$ as the replacement where the corresponding automaton is discharged from its boxes and their incoming and outgoing transitions.

Definition 4.2.8 (Sequential composition)**.** *Given an IBA* $\mathcal{M} = \langle \Sigma_\mathcal{M}, R_\mathcal{M}, B_\mathcal{M}, Q_\mathcal{M}, \Delta_\mathcal{M}, Q^0_\mathcal{M}, F_\mathcal{M} \rangle$ *and the replacement* $\mathcal{R}_b = \langle \mathcal{M}_b, \Delta^{inR_b}, \Delta^{outR_b} \rangle$ *of the box* $b \in B_\mathcal{M}$*, the sequential composition* $\mathcal{M} \bowtie \mathcal{R}_b$ *is an IBA* $\langle \Sigma_{\mathcal{M}\bowtie\mathcal{R}_b}, R_{\mathcal{M}\bowtie\mathcal{R}_b}, B_{\mathcal{M}\bowtie\mathcal{R}_b}, Q_{\mathcal{M}\bowtie\mathcal{R}_b}, \Delta_{\mathcal{M}\bowtie\mathcal{R}_b}, Q^0_{\mathcal{M}\bowtie\mathcal{R}_b}, F_{\mathcal{M}\bowtie\mathcal{R}_b} \rangle$ *of* $\mathcal{M}$ *that satisfies the following conditions:*

1. $\Sigma_{\mathcal{M}\bowtie\mathcal{R}_b} = \Sigma_\mathcal{M} \cup \Sigma_{\mathcal{M}_b}$;
2. $R_{\mathcal{M}\bowtie\mathcal{R}_b} = R_\mathcal{M} \cup R_{\mathcal{M}_b}$;
3. $B_{\mathcal{M}\bowtie\mathcal{R}_b} = B_\mathcal{M} \setminus \{b\} \cup B_{\mathcal{M}_b}$;

4. $Q_{\mathcal{M}\bowtie\mathcal{R}_b} = R_{\mathcal{M}\bowtie\mathcal{R}_b} \cup B_{\mathcal{M}\bowtie\mathcal{R}_b}$;
5. $\Delta_{\mathcal{M}\bowtie\mathcal{R}_b} = (\Delta_{\mathcal{M}} \setminus \{(q_{\mathcal{M}}, a, q'_{\mathcal{M}}) \in \Delta_{\mathcal{M}} \mid q_{\mathcal{M}} = b \vee q'_{\mathcal{M}} = b\}) \cup \Delta_{\mathcal{M}_b} \cup \Delta^{inR_b} \cup \Delta^{outR_b}$;
6. $Q^0_{\mathcal{M}\bowtie\mathcal{R}_b} = (Q^0_{\mathcal{M}} \cup Q^0_{\mathcal{M}_b}) \cap Q_{\mathcal{M}\bowtie\mathcal{R}_b}$;
7. $F_{\mathcal{M}\bowtie\mathcal{R}_b} = (F_{\mathcal{M}} \cup F_{\mathcal{M}_b}) \cap Q_{\mathcal{M}\bowtie\mathcal{R}_b}$.

Definition 4.2.8, condition 1, specifies that the alphabet of the refinement $\mathcal{M} \bowtie \mathcal{R}_b$ is the union of the alphabet of the original IBA $\mathcal{M}$ and the alphabet of its replacement $\mathcal{R}_b$. Definition 4.2.8, condition 2, specifies that the set of regular states of $\mathcal{M} \bowtie \mathcal{R}_b$ is the union of the set of the regular states of $\mathcal{M}$ and the set of the regular states of the replacement $\mathcal{R}_b$. Definition 4.2.8, condition 3, specifies that the set of boxes of $\mathcal{M} \bowtie \mathcal{R}_b$ is the union of the set of the boxes of $\mathcal{M}$, with the exception of the box b which is refined, and the set of the boxes of the replacement $\mathcal{R}_b$. Definition 4.2.8, condition 4, specifies the set of the states of $\mathcal{M} \bowtie \mathcal{R}_b$ which corresponds to the union of its regular and box states. Note that the box b is not contained into $Q_{\mathcal{M}\bowtie\mathcal{R}_b}$. Definition 4.2.8, condition 5, specifies the set of the transitions of $\mathcal{M} \bowtie \mathcal{R}_b$. The transitions include all the transitions of the original model $\Delta_{\mathcal{M}}$ with the exception of the transitions that reach and leaves the box b, all the transitions $\Delta_{\mathcal{M}_b}$ of the automaton that corresponds to the replacement and its incoming and outgoing transitions Δ^{inR_b} and Δ^{outR_b}. The set $Q^0_{\mathcal{M}\bowtie\mathcal{R}_b}$ of the initial states of $\mathcal{M} \bowtie \mathcal{R}_b$ includes all the initial states $Q^0_{\mathcal{M}}$ of the IBA and the initial states $Q^0_{\mathcal{M}_b}$ of its replacement. The intersection with the set $Q_{\mathcal{M}\bowtie\mathcal{R}_b}$ is computed to remove the box b (if present). The set $F_{\mathcal{M}\bowtie\mathcal{R}_b}$ of the accepting states of $\mathcal{M} \bowtie \mathcal{R}_b$ include all the accepting states $F_{\mathcal{M}}$ of the IBA and the accepting states $F_{\mathcal{M}_b}$ of its replacement. As previously, the intersection with the set $Q_{\mathcal{M}\bowtie\mathcal{R}_b}$ removes the box b (if present).

Theorem 4.2.2 (Refinement Preservation)**.** *Given a model* $\mathcal{M} = \langle \Sigma_{\mathcal{M}}, R_{\mathcal{M}}, B_{\mathcal{M}}, Q_{\mathcal{M}}, \Delta_{\mathcal{M}}, Q^0_{\mathcal{M}}, F_{\mathcal{M}} \rangle$ *and a replacement* $\mathcal{R}_b = \langle \mathcal{M}_b, \Delta^{inR_b}, \Delta^{outR_b} \rangle$ *which refers to one of its boxes* b*,* $\mathcal{M} \preceq \mathcal{M} \bowtie \mathcal{R}_b$*.*

Proof. To prove that $\mathcal{M} \preceq \mathcal{M} \bowtie \mathcal{R}_b$ we must define a refinement relation $\Re$ which satisfies the conditions specified in Definition 4.2.1.

The set of initial states $Q^0_{\mathcal{M}\bowtie\mathcal{R}_b}$ contains the initial states of $\mathcal{M}$ (with the exception of the refined box b) and the initial states of the automaton corresponding to replacement $\mathcal{R}_b$. It is possible to associate to each initial state of $\mathcal{M}$ (with the exception of the refined box b) the corresponding state of $\mathcal{M}$ and to each initial state of the replacement $\mathcal{R}_b$ the box b. Note that a replacement $\mathcal{R}_b$ can contain an initial state only if b is initial for $\mathcal{M}$. This construction guarantees that the relation $\Re$ satisfies the conditions 3 and 6 of the Definition 4.2.1. Conditions 4, 5 and 7 can be satisfied in a similar way, i.e., by associating the box/final state of $\mathcal{M} \preceq \mathcal{M} \bowtie \mathcal{R}_b$ to the corresponding state of the model or the box b that is refined. Let us finally analyze conditions 8 and 9. Each transition $\Delta_{\mathcal{M}}$ whose destination is not a box can be associated with the corresponding transition of the model, which makes 8 trivially satisfied. The transitions whose destinations are the box b can be associated with the corresponding transitions in Δ^{inR_b}. Note that Definition 4.2.3 forces each incoming/outgoing transition of a box to have at least a corresponding incoming/outgoing transition inside the replacement. Let us finally

consider the outgoing transition of the box b of $\mathcal{M}$. Each outgoing transition can be associated with the corresponding outgoing transition in Δ^{outR_b}. The same procedure can be applied to satisfy the condition 9. Note that, each transition in $\Delta_{\mathcal{M}_b}$ is associated with the box b. By following this procedure the refinement relation $\Re$ satisfies the conditions specified in Definition 4.2.1 by construction, therefore $\mathcal{M} \preceq \mathcal{M} \bowtie \mathcal{R}_b$ is satisfied. □

4.3 Modeling the claim

When a system is incomplete, a different semantic for the formulae of interest must be considered, e.g., a *three value* semantic. Given a formula ϕ (expressed in some logic) and an IBA $\mathcal{M}$ three truth values can be associated to the satisfaction of the formula ϕ in the model $\mathcal{M}$: *true*, *false* and *unknown* (maybe). Whenever a formula is true or false its satisfaction does not depend on the incomplete parts present in the model $\mathcal{M}$. In the fist case, all the behaviors of the system (including the one that the system may exhibit) satisfy the formula ϕ. In the second case, there exists a behavior of $\mathcal{M}$, which does not depend on the incomplete parts which violates the property of interest. In the third case, the satisfaction of ϕ depends on the incomplete parts. The three value semantic of LTL describes the semantic of LTL formulae over IBA.

4.3.1 Three value Linear Time Temporal Logic semantic

Given an LTL formula ϕ and an IBA model $\mathcal{M}$ the semantic function $\|\mathcal{M}^{\phi}\|$ associates to $\mathcal{M}$ and ϕ one of the true values true (T), false (F) and unknown ($\perp$). Whenever a formula is true, it is true in all the implementations of $\mathcal{M}$, i.e., it does not exists any refinement of the boxes that makes ϕ violated. If the formula is false, there exists a behavior of $\mathcal{M}$, which does not depend on how the system is refined which violates the property of interest. Thus, all the implementations of $\mathcal{M}$ will make ϕ not satisfied. In the third case, the satisfaction of ϕ depends on the refinement of the boxes of $\mathcal{M}$. This type of three value semantic is also known in literature as *inductive semantic* [141] and is different from the thorough semantic defined in [13].

Definition 4.3.1 (Three value LTL semantic over IBA)**.** *Given an IBA $\mathcal{M}$ and the claim ϕ:*

1. $\|\mathcal{M}^{\phi}\| = T$ *if and only if for all* $v \in (\mathcal{L}^{\omega}(\mathcal{M}) \cup \mathcal{L}_p^{\omega}(\mathcal{M}))$, $v \models \phi$
2. $\|\mathcal{M}^{\phi}\| = F$ *if and only if exists* $v \in \mathcal{L}^{\omega}(\mathcal{M})$ *such that* $v \not\models \phi$
3. $\|\mathcal{M}^{\phi}\| = \perp$ *if and only for all* $v \in \mathcal{L}^{\omega}(\mathcal{M})$, $v \models \phi$ *and there exists* $u \in \mathcal{L}_p^{\omega}(\mathcal{M})$ *such that* $u \not\models \phi$

A formula ϕ is true in the model $\mathcal{M}$ if and only if every word v that is in the language definitely accepted or possibly accepted by the automaton satisfies the claim ϕ (Definition 4.3.1, condition 1). A formula ϕ is false in the model $\mathcal{M}$ if and only if there exists word v that is in the language definitely accepted by the BA that does not satisfy the claim ϕ (Definition 4.3.1, condition 2). A formula ϕ is possibly satisfied in the model $\mathcal{M}$ if and only if there exists word u that is in the language possibly accepted by the BA that does not satisfy the claim ϕ, but all the words v in the language definitely

accepted by $\mathcal{M}$ satisfy the formula ϕ (Definition 4.3.1, condition 3). For example, the property $\phi = \Box(send \rightarrow \Diamond success)$ is possibly satisfied by the model described in Figure 4.1 since there exists a word $\{start\}.\{send\}.\{fail\}.\{fail\}.\{abort\}^{\omega}$ in the possibly accepted language which does not satisfy the formula and there are no words in the definitely accepted language.

Theorem 4.3.1 (Refinement preservation of LTL properties)**.** *Given an IBA $\mathcal{M}$ and its refinement $\mathcal{N}$, such that $\mathcal{M} \preceq \mathcal{N}$, Then:*

1. *if $\|\mathcal{M}^{\phi}\| = T$ then $\|\mathcal{N}^{\phi}\| = T$;*
2. *if $\|\mathcal{M}^{\phi}\| = F$ then $\|\mathcal{N}^{\phi}\| = F$.*

Proof. Let us first consider condition 1. The proof is done by contradiction. Assume that $\|\mathcal{M}^{\phi}\| = T$ and $\|\mathcal{N}^{\phi}\| \neq T$. If $\|\mathcal{N}^{\phi}\| = F$, by Definition 4.3.1 exists $v \in \mathcal{L}^{\omega}(\mathcal{N})$ such that $v \not\models \phi$. By Theorem 4.2.1 v must be in the possibly recognized language of $\mathcal{M}$, or neither in the possibly recognized nor in the definitely recognized language of $\mathcal{M}$. This implies that the condition 1 of the Definition 4.3.1 is not met, and the hypothesis $\|\mathcal{M}^{\phi}\| = T$ is contradicted. If $\|\mathcal{N}^{\phi}\| = \bot$, by Definition 4.3.1 it must exists a word $v \in \mathcal{L}_p^{\omega}(\mathcal{N})$, $v \not\models \phi$. By Theorem 4.2.1 $v \in \mathcal{L}_p^{\omega}(\mathcal{M})$, i.e., v must be in the possibly recognized language of $\mathcal{M}$. As previously, this implies that the condition 1 of the Definition 4.3.1 is not satisfied, i.e., it exists a word that does not satisfy ϕ and is possibly recognized by $\mathcal{M}$. Thus the hypothesis $\|\mathcal{M}^{\phi}\| = T$ is contradicted.

Let us now consider condition 2. Since $\|\mathcal{M}^{\phi}\| = F$ from Definition 4.3.1 condition 2 it must exists a word $v \in \mathcal{L}^{\omega}(\mathcal{M})$ that does not satisfy ϕ. By Definition 4.2.1 condition 1 $v \in \mathcal{L}^{\omega}(\mathcal{N})$. By Definition 4.3.1 Condition 2 we can conclude that $\|\mathcal{N}^{\phi}\| = F$. □

4.3.2 Three value Büchi Automata semantic

Given a BA $\mathcal{A}_{\phi}$ and an IBA $\mathcal{M}$ which describes the model of the system, the semantic function $\|\mathcal{M}^{\mathcal{A}_{\phi}}\|$ associates to the model $\mathcal{M}$ and the property $\mathcal{A}_{\phi}$ one of true values true T, false F and unknown $\bot$ depending on whether the model satisfies, possibly satisfies or does not satisfy the claim specified by the BA $\mathcal{A}_{\phi}$.

Definition 4.3.2 (Three value BA semantic)**.** *Given and incomplete BA $\mathcal{M}$ and a BA $\mathcal{A}_{\phi}$ which specifies the definitely accepted behaviors of $\mathcal{M}$,*

1. *$\|\mathcal{M}^{\mathcal{A}_{\phi}}\| = T$ iff $\mathcal{L}^{\omega}(\mathcal{M}) \cup \mathcal{L}_p^{\omega}(\mathcal{M}) \subseteq \mathcal{L}(\mathcal{A}_{\phi})$;*
2. *$\|\mathcal{M}^{\mathcal{A}_{\phi}}\| = \bot$ iff $\mathcal{L}^{\omega}(\mathcal{M}) \subseteq \mathcal{L}^{\omega}(\mathcal{A}_{\phi})$ and $\mathcal{L}_p^{\omega}(\mathcal{M}) \not\subseteq \mathcal{L}^{\omega}(\mathcal{A}_{\phi})$*
3. *$\|\mathcal{M}^{\mathcal{A}_{\phi}}\| = F$ iff $\mathcal{L}^{\omega}(\mathcal{M}) \not\subseteq \mathcal{L}^{\omega}(\mathcal{A}_{\phi})$*

Informally, a model $\mathcal{M}$ *satisfies* the claim expressed as a BA $\mathcal{A}_{\phi}$ if and only if the condition 1 is satisfied, i.e., all the behaviors of the model of the system, including possible behaviors, are contained in the set of the behaviors allowed by the property. A model $\mathcal{M}$ *possibly satisfies* the claim expressed as a BA $\mathcal{A}_{\phi}$ if and only if the condition 2 is satisfied, i.e., all the behaviors of the model of the system are contained into the set of behaviors allowed by the property but there exists a possible behavior which is not contained into the set of behaviors allowed by the property. Finally, condition 3

specifies that a model $\mathcal{M}$ *does not satisfy* the claim expressed as a BA $\mathcal{A}_\phi$ if and only if there exists a behavior of the model which is not allowed by the property.

Lemma 4.3.1 (Relation between Automata Based LTL Semantic)**.** *Given an LTL formula ϕ and the corresponding BA $\mathcal{A}_\phi$, $\|\mathcal{M}^\phi\| = \|\mathcal{M}^{\mathcal{A}_\phi}\|$.*

Proof. The proof follows from the fact that the automaton $\mathcal{A}_\phi$ contains all the words that satisfy the claim ϕ. Thus, asking for language containment as done in Definition 4.3.2 corresponds with checking that all the words accepted and possibly accepted by $\mathcal{M}$ satisfy the claim ϕ as done in Definitions 4.3.1. □

Lemma 4.3.1 allows relating the satisfaction of LTL formulae with respect to IBAs and is necessary since the models and claims of interest must have compatible semantics [33].

CHAPTER 5

Reasoning on Incomplete Systems

"You got to have a problem that you want to solve, a wrong that you want to right, it's got to be something that you're passionate about because otherwise you won't have the perseverance to see it through."

Steven Paul Jobs, 1955-2011

The core of the envisaged development process is the *development-analysis* cycle. During the *development* phase, designers refine an incomplete model $\mathcal{M}$ which describes the system up to some level of abstraction. At each development step, they produce a new replacement (increment) which describes the behavior of the system inside one of its black box states, leading to a new refined model $\mathcal{N}$, which may in turn contain incompleteness. When an increment is ready, developers analyze the properties of the refined model $\mathcal{N}$. If the model satisfies the designer's expectation, the development-analysis cycle is repeated, i.e., the development of the new increment is started.

The verification of incomplete models offers three major benefits: *a*) instead of forcing the verification procedure to be performed at the end of the development process, it allows the system to be checked at the early stages of the design; *b*) complex parts of the design can be encapsulated into unspecified (incomplete) parts (*abstraction*); *c*) the location of design errors can be identified by sequentially narrowing portions of the system into incomplete parts. The incomplete model checking algorithm proposed to check if an LTL property ϕ is satisfied, possibly satisfied or not satisfied by an IBA $\mathcal{M}$ that describes the system behaviors is described in Section 5.1.

If the property is *possibly* satisfied, the developer may be interested in the possible ways in which the boxes can be refined without violating the requirements of interest, i.e., the constraint to be followed in the refinement of incompleteness. The constraint is a set of sub-properties, one for each box. Each sub-property contains an automaton

that specifies the behaviors that can be exhibited by the replacement of the box. The constraint computation algorithm proposed in this thesis is described in Section 5.2.

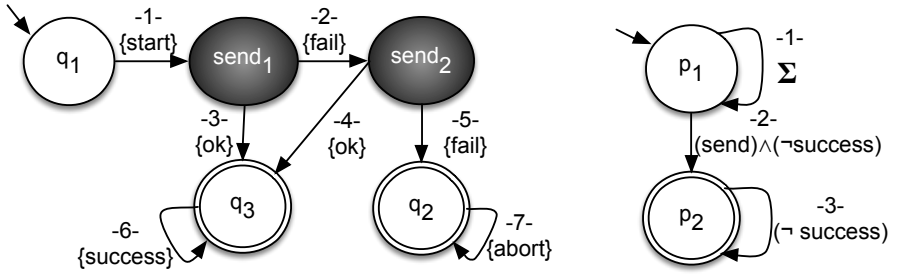
Whenever a new replacement $\mathcal{R}_b$ for the box b is proposed, it is important to guarantee that the new system satisfies the properties of interest. One of the possible ways to perform this operation is to inject the replacement $\mathcal{R}_b$ inside the original model $\mathcal{M}$, obtaining its refinement $\mathcal{N}$, and check the refinement against the property ϕ. However, at each refinement round, it is desirable not to verify the whole model from scratch, but to perform the verification in an incremental way. For this reason, Section 5.3 describes a model checking procedure which is able to consider the replacement $\mathcal{R}_b$ against the corresponding sub-property obtained from the previously computed constraint.

5.1 Checking incomplete Büchi Automata

Given an IBA $\mathcal{M}$ and an LTL formula ϕ, the incomplete model checking problem verifies whether the model satisfies, possibly satisfies or does not satisfy the property ϕ, i.e., $\|\mathcal{M}^{\phi}\|$ is equal to true (T), false (F) or maybe ($\perp$). Given an LTL formula ϕ, it is possible to transform the formula into a corresponding BA $\mathcal{A}_\phi$ and check $\|\mathcal{M}^{\mathcal{A}_\phi}\|$. Since BAs are closed under intersection and complementation, it is possible to transform $\neg\phi$ into the corresponding automaton $\overline{\mathcal{A}_\phi}$ and to reformulate the Conditions 1, 2 and 3 of Definition 4.3.2 as: $(\mathcal{L}(\mathcal{M}) \cup \mathcal{L}_p(\mathcal{M})) \cap \mathcal{L}(\overline{\mathcal{A}_\phi}) = \emptyset$; $\mathcal{L}(\mathcal{M}) \cap \mathcal{L}(\overline{\mathcal{A}_\phi}) = \emptyset$ and $\mathcal{L}_p(\mathcal{M}) \cap \mathcal{L}(\overline{\mathcal{A}_\phi}) \neq \emptyset$; and $\mathcal{L}(\mathcal{M}) \cap \mathcal{L}(\overline{\mathcal{A}_\phi}) \neq \emptyset$, respectively. However, to check these conditions, it is necessary to redefine the behavior of the intersection operator ($\cap$) over an IBA and a BA.

5.1.1 The intersection automaton

This section describes how the intersection between an IBA and a BA is computed. To exemplify the intersection between an IBA and a BA we will consider the model $\mathcal{M}$ presented in Figure 5.1a and the automaton corresponding to the negation of the LTL claim $\phi = G(send \rightarrow F(success))$ represented in Figure 5.1b.



(a) *The IBA that corresponds to model $\mathcal{M}$.*

(b) *The BA $\overline{\mathcal{A}_\phi}$ that represents $\neg\phi$.*

Figure 5.1: *The IBA and the BA used as examples in the description of the computation of the intersection automaton $\mathcal{I}$.*

Definition 5.1.1 (Intersection between a BA and an IBA)**.** *The intersection automaton $\mathcal{I} = \mathcal{M} \cap \overline{\mathcal{A}_\phi}$ between an IBA $\mathcal{M}$ and a BA $\overline{\mathcal{A}_\phi}$ is the BA $\mathcal{I} = \langle \Sigma_\mathcal{I}, Q_\mathcal{I}, \Delta_\mathcal{I}, Q^0_\mathcal{I}, F_\mathcal{I} \rangle$, such as:*

- $\Sigma_{\mathcal{I}} = \Sigma_{\mathcal{M}} \cup \Sigma_{\overline{\mathcal{A}_\phi}}$ *is the alphabet of* $\mathcal{I}$*;*
- $Q_{\mathcal{I}} = ((R_{\mathcal{M}} \times R_{\overline{\mathcal{A}_\phi}}) \cup (B_{\mathcal{M}} \times R_{\overline{\mathcal{A}_\phi}})) \times \{0,\ 1,\ 2\}$ *is the set of states;*
- $\Delta_{\mathcal{I}} = \Delta^c_{\mathcal{I}} \cup \Delta^p_{\mathcal{I}}$ *is the set of transitions of the intersection automaton.* $\Delta^c_{\mathcal{I}}$ *is the set of transitions* $(\langle q_i, q'_j, x\rangle, a,\ \langle q_m, q'_n, y\rangle)$*, such that* $(q_i, a, q_m) \in \Delta_{\mathcal{M}}$ *and* $(q'_j, a, q'_n) \in \Delta_{\overline{\mathcal{A}_\phi}}$*.* $\Delta^p_{\mathcal{I}}$ *corresponds to the set of transitions* $(\langle q_i, q'_j, x\rangle, a,\ \langle q_m, q'_n, y\rangle)$ *where* $q_i = q_m$ *and* $q_i \in B_{\mathcal{M}}$ *and* $(q'_j, a, q'_n) \in \Delta_{\overline{\mathcal{A}_\phi}}$*. Moreover, each transition in* $\Delta_{\mathcal{I}}$ *must satisfy the following conditions:*
  - *if* $x = 0$ *and* $q_m \in F_{\mathcal{M}}$*, then* $y = 1$*;*
  - *if* $x = 1$ *and* $q'_n \in F_{\overline{\mathcal{A}_\phi}}$*, then* $y = 2$*;*
  - *if* $x = 2$*, then* $y = 0$*;*
  - *otherwise,* $y = x$*;*
- $Q^0_{\mathcal{I}} = Q^0_{\mathcal{M}} \times Q^0_{\overline{\mathcal{A}_\phi}} \times \{0\}$ *is the set of initial states;*
- $F_{\mathcal{I}} = F_{\mathcal{M}} \times F_{\overline{\mathcal{A}_\phi}} \times \{2\}$ *is the set of accepting states.*

The intersection $\mathcal{I}$ between the model $\mathcal{M}$, depicted in Figure 5.1a, and the BA $\overline{\mathcal{A}_\phi}$ of Figure 5.1b, that corresponds to the negation of the property, is the BA described in Figure 5.2. The portions of the state space that contain mixed states associated with the black box states of the model $send_1$ and $send_2$ are surrounded by a dashed-dotted frame.

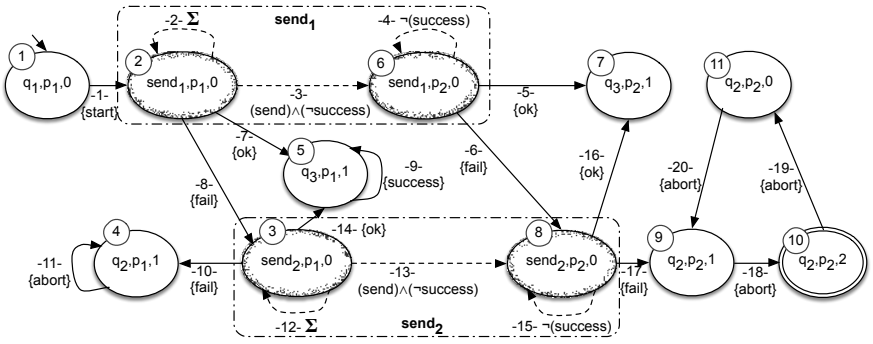


Figure 5.2: *The intersection automaton* $\mathcal{I}$ *between the incomplete BA* $\mathcal{M}$ *and the BA automaton* $\overline{\mathcal{A}_\phi}$ *which corresponds to the negation of the property* ϕ*.*

The alphabet $\Sigma_{\mathcal{I}}$ includes all the characters of the alphabets of $\mathcal{M}$ and $\overline{\mathcal{A}_\phi}$. The set $Q_{\mathcal{I}}$ is composed by the states obtained combining states of the automaton associated with the negation of the property $\overline{\mathcal{A}_\phi}$ with regular states and boxes of the model $\mathcal{M}$. As in the classical intersection algorithm for BAs [33], the labels 0, 1 and 2 indicate that no accepting state is entered, at least one accepting state of $\mathcal{M}$ is entered, and at least one accepting state of $\mathcal{M}$ and one accepting state of $\overline{\mathcal{A}_\phi}$ are entered, respectively. We define $M_{\mathcal{I}} = B_{\mathcal{M}} \times R_{\overline{\mathcal{A}_\phi}} \times \{0,\ 1,\ 2\}$ as the set of *mixed* states (graphically indicated in Figure 5.1b with a stipple border), and $PR_{\mathcal{I}} = R_{\mathcal{M}} \times R_{\overline{\mathcal{A}_\phi}} \times \{0,\ 1,\ 2\}$ as the set of *purely regular* states. For example, state ① is obtained by combining the state q_1 of

$\mathcal{M}$ and p_1 of $\overline{\mathcal{A}_\phi}$. This state is initial and purely regular since both q_1 and p_1 are initials and regulars. Conversely, state ② is mixed, since it is obtained by combining the box $send_1$ of $\mathcal{M}$ and the regular state p_1 of $\overline{\mathcal{A}_\phi}$.

The transitions in $\Delta^c_{\mathcal{I}}$ are obtained by the synchronous execution of the transitions of $\mathcal{M}$ and the transitions of $\overline{\mathcal{A}_\phi}$. For example, the transition from ② to ③ is obtained by combining the transition from $send_1$ to $send_2$ of $\mathcal{M}$ and the transition from p_1 to p_1 of $\overline{\mathcal{A}_\phi}$. The transitions in $\Delta^p_{\mathcal{I}}$ are, instead, obtained when a transition of $\overline{\mathcal{A}_\phi}$ synchronizes with a transition in the refinement of a box of $\mathcal{M}$. For example, the transition from ② to ⑥ is performed when $\overline{\mathcal{A}_\phi}$ moves from p_1 to p_2 and the automaton $\mathcal{M}$ performs a transition in the refinement of the box $send_1$.

The language recognized by $\mathcal{I}$ is the intersection of the language possibly recognized and recognized by $\mathcal{M}$ and the language recognized by $\overline{\mathcal{A}_\phi}$. For example, the word $\{start\}.\{send\}.\{fail\}.\{fail\}.\{abort\}^\omega$ is possibly recognized by $\mathcal{M}$ and is recognized by $\overline{\mathcal{A}_\phi}$. Indeed, the system may fire the $start$ transition, then perform a transition in the refinement of the state $send_1$ which satisfies the condition $(send) \wedge (\neg success)$, then perform the two $fail$ transitions and finally perform the transition labeled with $abort$ an infinite number of times.

Proposition 5.1.1 (Size of the intersection automaton)**.** *The intersection automaton $\mathcal{I}$ contains in the worst case $3 \cdot |Q_{\mathcal{M}}| \cdot |Q_{\overline{\mathcal{A}_\phi}}|$ states and can be computed in $\mathcal{O}(|\mathcal{M}| \cdot |\overline{\mathcal{A}_\phi}|)$.*

Lemma 5.1.1 (Intersection language)**.** *The intersection automaton $\mathcal{I} = \mathcal{M} \cap \overline{\mathcal{A}_\phi}$ between an incomplete BA $\mathcal{M}$ and a BA $\overline{\mathcal{A}_\phi}$ recognizes the language $(\mathcal{L}^\omega(\mathcal{M}) \cup \mathcal{L}^\omega_p(\mathcal{M})) \cap \mathcal{L}^\omega(\overline{\mathcal{A}_\phi})$, i.e., $v \in \mathcal{L}(\mathcal{I}) \Leftrightarrow v \in (\mathcal{L}^\omega(\mathcal{M}) \cup \mathcal{L}^\omega_p(\mathcal{M})) \cap \mathcal{L}^\omega(\overline{\mathcal{A}_\phi})$.*

Proof. ($\Rightarrow$) If $v \in \mathcal{L}(\mathcal{I})$, it must exists an accepting run ρ^ω in the intersection automaton which recognizes v. Since, by Definition 3.1.7, $\mathcal{I}$ is a BA for all $i > 0$ $\rho^\omega(i)$ and $\rho^\omega(i+1)$ are states of ρ^ω if and only if $(\rho^\omega(i), v_i, \rho^\omega(i+1)) \in \Delta_{\mathcal{I}}$. Let us consider the two states of the model $\rho^\omega_{\mathcal{M}}(i)$ and $\rho^\omega_{\mathcal{M}}(i+1)$ associated with $\rho^\omega(i)$ and $\rho^\omega(i+1)$. Since there exists a transition $(\rho^\omega(i), v_i, \rho^\omega(i+1)) \in \Delta_{\mathcal{I}}$, by construction it must exists a transition $(\rho^\omega_{\mathcal{M}}(i), v_i, \rho^\omega_{\mathcal{M}}(i+1)) \in \Delta_{\mathcal{M}}$ or $\rho^\omega_{\mathcal{M}}(i) = \rho^\omega_{\mathcal{M}}(i+1) \in B_{\mathcal{M}}$. In the first case, v_i is recognized by a transition of the model, in the second case, it is recognized by a box. Since this condition must hold $\forall i \geq 0$ it follows that $v \in (\mathcal{L}^\omega(\mathcal{M}) \cup \mathcal{L}^\omega_p(\mathcal{M}))$. The same idea can be applied with respect to the automaton $\overline{\mathcal{A}_\phi}$, which implies that $v \in \mathcal{L}(\overline{\mathcal{A}_\phi})$. Thus, $v \in (\mathcal{L}^\omega(\mathcal{M}) \cup \mathcal{L}^\omega_p(\mathcal{M})) \cap \mathcal{L}^\omega(\overline{\mathcal{A}_\phi})$.

($\Leftarrow$) The proof is by contradiction. Imagine that there exists a word $v \notin \mathcal{L}(\mathcal{I})$ which is in $(\mathcal{L}^\omega(\mathcal{M}) \cup \mathcal{L}^\omega_p(\mathcal{M})) \cap \mathcal{L}^\omega(\overline{\mathcal{A}_\phi})$. Since $v \notin \mathcal{L}(\mathcal{I})$, it is not recognized by $\mathcal{L}(\mathcal{I})$, i.e., for every possible accepting run ρ^ω it must exists a character v_i of v such that $(\rho^\omega(i), v_i, \rho^\omega(i+1)) \notin \Delta_{\mathcal{I}}$. Let us consider the corresponding states $\rho^\omega_{\mathcal{M}}(i)$, $\rho^\omega_{\mathcal{M}}(i+1)$ and $\rho^\omega_{\overline{\mathcal{A}_\phi}}(i)$, $\rho^\omega_{\overline{\mathcal{A}_\phi}}(i+1)$ of the model and of the claim, respectively. To make $(\rho^\omega(i), v_i, \rho^\omega(i+1)) \notin \Delta_{\mathcal{I}}$ two cases are possible: $(\rho^\omega_{\overline{\mathcal{A}_\phi}}(i), v_i, \rho^\omega_{\overline{\mathcal{A}_\phi}}(i+1)) \notin \Delta_{\overline{\mathcal{A}_\phi}}$ or $(\rho^\omega_{\mathcal{M}}(i), v_i, \rho^\omega_{\mathcal{M}}(i+1)) \notin \Delta_{\mathcal{M}}$ and not $\rho^\omega(i)_{\mathcal{M}} = \rho^\omega(i+1)_{\mathcal{M}} \in B_{\mathcal{M}}$. However, in the first case, the condition implies that $v \not in \mathcal{L}^\omega(\overline{\mathcal{A}_\phi}$, while in the second, $v \notin (\mathcal{L}^\omega(\mathcal{M}) \cup \mathcal{L}^\omega_p(\mathcal{M}))$, thus $v \notin (\mathcal{L}^\omega(\mathcal{M}) \cup \mathcal{L}^\omega_p(\mathcal{M})) \cap \mathcal{L}^\omega(\overline{\mathcal{A}_\phi})$, contradicting the hypothesis. □

Given an infinite run ρ^ω of $\mathcal{I}$ associated with the infinite word $v \in \mathcal{L}(\mathcal{I})$, we may want to identify the portions of the word v which are recognized by each box $b \in B_{\mathcal{M}}$.

Note that these portions may include both finite words (i.e., finite portions of the words that are recognized inside the boxes) or infinite words (i.e., suffixes of words recognized inside *accepting* boxes).

Definition 5.1.2 (Finite abstractions of a run)**.** *Given an infinite run ρ^ω of $\mathcal{I} = \mathcal{M} \cap \overline{\mathcal{A}_\phi}$ associated with the infinite word $v \in \mathcal{L}^\omega(\mathcal{I})$ and a box $b \in B_\mathcal{M}$, $\alpha_b^*(v, \rho^\omega)$ is the set of* finite *words $\nu_{init}.\nu.\nu_{out} \in \Sigma^*$ associated with the box b and the run ρ^ω of the infinite word v. A word $\nu^* = \nu_{init}.\nu.\nu_{out}$ is in $\alpha_b^*(v, \rho^\omega)$ if and only if given two indexes i, j such that $0 \leq i < j < \infty$, for all $0 \leq k < j - i$ the following conditions must be satisfied:*

1. $\nu_k^* = v_{i+k}$;
2. $\rho^\omega(i + k) = \langle b, p, x\rangle$;
3. $(\rho(k)^\omega, v_{i+k}, \rho^\omega(k + 1)) \in \Delta_i^p$;
4. $(\rho(j)^\omega, v_j, \rho^\omega(j + 1)) \in \Delta_i^c$ *and* $\nu_{out} = v_j$;
5. $(i > 0 \Leftrightarrow (\rho^\omega(i - 1), v_{i-1}, \rho^\omega(i)) \in \Delta_i^c$ *and* $\nu_{init} = v_{i-1})$ *or* $\nu_{init} = \epsilon$.

Condition 1 specifies that ν^* contains only the portion of the word of interest, i.e., the portion of the word recognized by the box. Condition 2 specifies that the state $\rho^\omega(i+k)$ of the run must corresponds to the tuple $\langle b, p, x\rangle$ where b is the box of interest. Condition 3 specifies that the transition that recognizes the character v_{i+k} must be in Δ_i^p, i.e., it is obtained by firing a transition of the claim when the system is inside the box b. Condition 4 forces the word to be of maximal length, i.e., the transition $(\rho^\omega(j), v_j, \rho^\omega(j + 1))$ of the run ρ^ω must be a transition that forces the model to leave the box b. The corresponding character v_j is added as a suffix ν_{out} of the word ν. Similarly, Condition 5 forces the transition that precedes the set of transitions which recognize ν to be a transition that enters the box b (excluding the case in which $i = 0$, i.e., the initial state of the run $\rho^\omega(0)$ is mixed, in which $\nu_{init} = \epsilon$[1]). The character v_{i-1} which labels the transition must be used as a prefix ν_{init} of ν.

Let us consider the intersection automaton $\mathcal{I}(\mathcal{M} \cap \overline{\mathcal{A}_\phi})$ presented in Figure 5.2, the word $v = \{start\}.\{send\}.\{fai\}^\omega$, and the corresponding run ρ^ω = ①②⑥⑧(⑨⑩⑪)$^\omega$. The finite abstraction of the run associated with the state $send_1$ is the function $\alpha^*_{send_1}$ such that, $\alpha^*_{send_1}(v, \rho^\omega) = \{start\}.\{send\}.\{fail\}$.

Definition 5.1.3 (Finite abstraction of the intersection)**.** *Given the intersection automaton $\mathcal{I} = \mathcal{M} \cap \overline{\mathcal{A}_\phi}$, and a box $b \in B_\mathcal{M}$, the set $\alpha_b^*(\mathcal{I})$ of the finite abstraction of the intersection automaton is defined as $\alpha_b^*(\mathcal{I}) = \{ \bigcup_{\forall v \in \mathcal{L}^\omega(\mathcal{I})} \alpha_b^*(v, \rho^\omega)$, such that ρ^ω is an accepting run of v $\}$.*

The finite abstraction of the intersection automaton contains the finite abstraction of the runs associated to every possible word in $\mathcal{L}^\omega(\mathcal{I})$.

Definition 5.1.4 (Infinite abstractions of a run)**.** *Given an infinite run ρ^ω of $\mathcal{I} = \mathcal{M} \cap \overline{\mathcal{A}_\phi}$ associated with the infinite word $v \in \mathcal{L}^\omega(\mathcal{I})$ and a box $b \in B_\mathcal{M}$, $\alpha_b^\omega(v, \rho^\omega)$ is the set of* infinite *words $\nu_{init}.\nu^\omega \in \Sigma^\omega$ associated with box b and the run ρ^ω of the infinite word v. A word $\nu^\omega = \nu_{init}.\nu$ is in $\alpha_b^\omega(v, \rho^\omega)$, if and only if given the index $i \geq 0$, $\forall 0 \leq k$ the following conditions are be satisfied:*

[1] The ϵ character denotes an empty string.

1. $\nu_k^\omega = v_{i+k}$;
2. $\rho^\omega(i+k) = \langle b, p, x \rangle$;
3. $(\rho^\omega(k), v_{i+k}, \rho^\omega(k+1)) \in \Delta_i^p$;
4. $(i > 0 \Leftrightarrow (\rho^\omega(i-1), v_{i-1}, \rho^\omega(i)) \in \Delta_i^c$ *and* $\nu_{init} = v_{i-1})$ *or* $\nu_{init} = \epsilon$.

Condition 1 specifies that ν^ω contains only the portion of the word of interest, i.e., the portion of the word recognized by the box b. Condition 2 specifies that the state $\rho^\omega(i+k)$ of the run must corresponds to the tuple $\langle b, p, x \rangle$ where b is the box of interest. Condition 3 specifies that the transition that recognizes the character v_{i+k} must be in Δ_i^p, i.e., it is obtained by firing a transition of the claim when the system is inside the box b. Condition 4 forces the transition that precedes the set of transitions which recognize ν to be a transition that enters the box b (excluding the case in which $i = 0$, i.e., the initial state of the run $\rho^\omega(0)$ is mixed, in which $\nu_{init} = \epsilon$). The character v_{i-1} which labels the transition must be used as a prefix ν_{init} of ν.

Definition 5.1.5 (Infinite abstraction of the intersection)**.** *Given the intersection automaton* $\mathcal{I} = \mathcal{M} \cap \overline{\mathcal{A}_\phi}$ *and a box* $b \in B_\mathcal{M}$*, the set* $\alpha_b^\omega(\mathcal{I})$ *of the infinite abstractions of the intersection automaton is defined as* $\alpha_b^\omega(\mathcal{I}) = \{ \bigcup_{\forall v \in \mathcal{L}^\omega(\mathcal{I})} \alpha_b^\omega(v, \rho^\omega)$ *such that* ρ^ω *is an accepting run of* $v \}$.

Informally, the infinite abstraction of the intersection automaton contains the infinite abstractions of the runs associated to every possible word recognized by the automaton $\mathcal{I}$.

5.1.2 The model checking procedure

The model checking procedure between an IBA $\mathcal{M}$ and a LTL property ϕ is based on the intersection between a BA and an IBA (Definition 5.1.1) and the completion of an IBA (Definition 4.1.13).

Definition 5.1.6 (Incomplete Model Checking)**.** *Given an IBA* $\mathcal{M}$ *and a LTL formula* ϕ *associated with the Büchi automaton* $\mathcal{A}_\phi$*,*

1. $\|\mathcal{M}^\phi\| = F \Leftrightarrow \mathcal{M}_c \cap \overline{\mathcal{A}_\phi} \neq \emptyset$;
2. $\|\mathcal{M}^\phi\| = \bot \Leftrightarrow \|\mathcal{M}^\phi\| \neq F$ *and* $\mathcal{M} \cap \overline{\mathcal{A}_\phi} \neq \emptyset$;
3. $\|\mathcal{M}^\phi\| = T \Leftrightarrow \|\mathcal{M}^\phi\| \neq F$ *and* $\|\mathcal{M}^\phi\| \neq \bot$.

Theorem 5.1.1 (Incomplete Model Checking correctness)**.** *The incomplete model checking technique is correct.*

Proof. Let us consider condition 1. ($\Leftarrow$) By Lemma 4.1.2, $\mathcal{M}_c$ recognizes the language $\mathcal{L}^\omega(\mathcal{M})$. Since $\overline{\mathcal{A}_\phi}$ describes all the possible infinite words that violate ϕ, if $\mathcal{M}_c \cap \overline{\mathcal{A}_\phi} \neq \emptyset$ it must exists a word $v \in \mathcal{L}^\omega(\mathcal{M})$ which violates ϕ. Therefore, by Definition 4.3.1 it follows that $\|\mathcal{M}^\phi\| = F$. ($\Rightarrow$) As specified in Lemma 4.3.1 $\|\mathcal{M}^\phi\| = F$ implies that there exist $v \in \mathcal{L}^\omega(\mathcal{M}), v \not\models \phi$. Since v does not model ϕ, it is a violating behavior and is included in $\mathcal{L}^\omega(\overline{\mathcal{A}_\phi})$. Since $v \in \mathcal{L}^\omega(\overline{\mathcal{A}_\phi})$ and $v \in \mathcal{L}^\omega(\mathcal{M})$ and from Lemma 4.1.2 $\mathcal{L}^\omega(\mathcal{M})$ is recognized by $\mathcal{M}_c$, $\mathcal{M}_c \cap \overline{\mathcal{A}_\phi}$ contains at least the word v.

Let us then consider condition 2. ($\Leftarrow$) Given that $\|\mathcal{M}^{\phi}\| \neq F$ by Definition 4.3.1 it follows that for all $v \in \mathcal{L}^{\omega}(\mathcal{M})$, the word v satisfies ϕ. Since $\mathcal{M} \cap \overline{\mathcal{A}_{\phi}} \neq \emptyset$, by Lemma 5.1.1 it must exists a word v such that $v \in (\mathcal{L}^{\omega}(\mathcal{M}) \cup \mathcal{L}_{p}^{\omega}(\mathcal{M})) \cap \mathcal{L}^{\omega}(\overline{\mathcal{A}_{\phi}})$. Since for all $v \in \mathcal{L}^{\omega}(\mathcal{M}), v \models \phi$, $\mathcal{L}^{\omega}(\mathcal{M}) \cap \mathcal{L}^{\omega}(\overline{\mathcal{A}_{\phi}}) = \emptyset$, it must be that $\mathcal{L}_{p}^{\omega}(\mathcal{M}) \cap \mathcal{L}^{\omega}(\overline{\mathcal{A}_{\phi}}) \neq \emptyset$. Thus, by Definition 4.3.2 condition 2 it must exist $u \in \mathcal{L}_{p}^{\omega}(\mathcal{M})$ such that $u \not\models \phi$. Thus, by Definition 4.3.1, condition 3 it must be that $\|\mathcal{M}^{\phi}\| = \bot$. ($\Rightarrow$) The proof is by contradiction. Imagine that $\|\mathcal{M}^{\phi}\| = \bot$ and $\|\mathcal{M}^{\phi}\| = F$ or $\mathcal{M} \cap \overline{\mathcal{A}_{\phi}} = \emptyset$. Let us consider the case in which $\|\mathcal{M}^{\phi}\| = F$. By Definition 4.3.1 condition 2 it must exists a word $v \in \mathcal{L}^{\omega}(\mathcal{M})$ that does not satisfy ϕ. This implies that condition 3 of Definition 4.3.1 is not satisfied and $\|\mathcal{M}^{\phi}\| \neq \bot$ making the hypothesis contradicted. Consider then the case in which $\mathcal{M} \cap \overline{\mathcal{A}_{\phi}} = \emptyset$. Since the intersection automaton $\mathcal{I} = \mathcal{M} \cap \overline{\mathcal{A}_{\phi}}$ is empty, from Lemma 5.1.1 it must not exists a word $v \in (\mathcal{L}^{\omega}(\mathcal{M}) \cup \mathcal{L}_{p}^{\omega}(\mathcal{M})) \cap \mathcal{L}^{\omega}(\overline{\mathcal{A}_{\phi}})$. This implies that condition 3 of Definition 4.3.1 is not satisfied and $\|\mathcal{M}^{\phi}\| \neq \bot$ making the hypothesis contradicted.

Condition 3 is a consequence of the previous conditions and Definition 4.3.1. □

Algorithm 1 Checks if an IBA satisfies a LTL property ϕ

```
1:  procedure MODELCHECKING(M, φ)
2:      Aφ ← LTL2BA(¬φ);
3:      Mc ← ExtractMc(M);
4:      Ic ← Mc ∩ Aφ;
5:      empty ← CheckEmptiness(Ic);
6:      if !empty then
7:          return F;
8:      else
9:          I(M ∩ Aφ) ← M ∩ Aφ;
10:         empty ← CheckEmptiness(I);
11:         if empty then
12:             return T;
13:         else
14:             return ⊥;
15:         end if
16:     end if
17: end procedure
```

Theorem 5.1.6 suggests the model checking procedure presented in Algorithm 1. The algorithm works in five different steps:

- *Create the automaton $\overline{\mathcal{A}_{\phi}}$ (Line 2).* As in the classical model checking framework, the first step is to construct the BA that contains the set of behaviors forbidden by the property ϕ. The temporal complexity of this step is $\mathcal{O}(2^{(|\neg\phi|)})$;

For example, the automaton $\overline{\mathcal{A}_{\phi}}$, corresponding to the LTL property $\phi = G(send \rightarrow F(success))$, is represented in Figure 5.1b.

- *Extract the automaton $\mathcal{M}_c$ and build the intersection automaton $\mathcal{I}_c = \mathcal{M}_c \cap \overline{\mathcal{A}_{\phi}}$ (Lines 3-4).* The automaton $\mathcal{M}_c$ contains all the accepting behaviors of the system. In this sense $\mathcal{M}_c$ is a lower bound on the set of behaviors of the system, i.e., it contains all the behaviors the system is going to exhibit at run-time. Thus, the

intersection automaton $\mathcal{I}_c$ contains the behaviors of $\mathcal{M}_c$ that violate the property. Computing $\mathcal{M}_c$ has in the worst case temporal complexity $\mathcal{O}(|Q_{\mathcal{M}}|+|\Delta_{\mathcal{M}}|)$ since it is sufficient to remove from the automaton the black box states and their incoming and outgoing transitions. The intersection automaton $\mathcal{I}_c$ contains in the worst case $3 \cdot |R_{\mathcal{M}}| \cdot |Q_{\overline{\mathcal{A}_\phi}}|$ states.

For example, the automaton $\mathcal{M}_c$ associated with the the model $\mathcal{M}$ described in Figure 5.1a contains only the states q_1, q_2 and q_3 and the transitions marked with the labels $success$ and $abort$. The intersection between this automaton and the property $\overline{\mathcal{A}_\phi}$ described in Figure 5.1b contains all the behaviors of the sending message system that violate the property. Since the automaton is empty, there are no behaviors of $\mathcal{M}$ that violate ϕ.

- *Check the emptiness of the intersection automaton $\mathcal{I}_c$ (Lines 5-8).* If $\mathcal{I}_c$ is not empty, the condition $\mathcal{L}(\mathcal{M}) \cap \mathcal{L}(\overline{\mathcal{A}_\phi}) \neq \emptyset$ introduced in c) is matched, i.e., the property is not satisfied and every infinite word in the intersection automaton is a counterexample. If, instead, $\mathcal{I}_c$ is empty, $\mathcal{M}$ possibly satisfies or satisfies ϕ depending on the result of the next steps of the algorithm.

The intersection automaton $\mathcal{I}_c$ of the sending message example is empty since the automaton $\mathcal{I}_c$ does not contain any accepting state reachable from the initial state. Indeed, both q_2 and q_3, which are the accepting states of $\mathcal{M}_c$ are never reachable from q_1 in the intersection automaton. Thus, $\mathcal{M}$ satisfies or possibly satisfies the property ϕ depending on the next steps of the algorithm.

- *Compute the intersection $\mathcal{I} = \mathcal{M} \cap \overline{\mathcal{A}_\phi}$ of the incomplete model $\mathcal{M}$ and the automaton $\overline{\mathcal{A}_\phi}$ associated with the property ϕ (Line 9).* To check whether $\mathcal{M}$ satisfies or possibly satisfies ϕ it is necessary to verify if $(\mathcal{L}(\mathcal{M}) \cup \mathcal{L}_p(\mathcal{M})) \cap \mathcal{L}(\overline{\mathcal{A}_\phi}) = \emptyset$, since $\mathcal{M}$ intrinsically specifies as an upper bound on the behaviors of the system, i.e., it contains all the behaviors the system must and may exhibit. The intersection algorithm presented in Section 5.1.1 is used to compute the intersection automaton $\mathcal{I} = \mathcal{M} \cap \overline{\mathcal{A}_\phi}$.

The intersection automaton $\mathcal{I}$ of the sending message example is depicted in Figure 5.2.

- *Check the emptiness of the intersection automaton $\mathcal{I} = \mathcal{M} \cap \overline{\mathcal{A}_\phi}$ (Lines 10-15).* By checking the emptiness of the automaton $\mathcal{I}$ we verify whether the property ϕ is satisfied or possibly satisfied by the model $\mathcal{M}$. Since we have already checked that $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\overline{\mathcal{A}_\phi})$, two cases are possible: if $\mathcal{I}$ is empty, $\mathcal{L}_p(\mathcal{M}) \subseteq \mathcal{L}(\overline{\mathcal{A}_\phi})$ and the property is satisfied whatever refinement is proposed for the boxes of $\mathcal{M}$, otherwise, $\mathcal{L}_p(\mathcal{M}) \not\subseteq \mathcal{L}(\overline{\mathcal{A}_\phi})$, meaning that there exists some refinement of $\mathcal{M}$ that violates the property.

For example, the word $\{start\}.\{send\}.\{fail\}.\{fail\}.\{abort\}^\omega$, which is a possibly accepted by $\mathcal{M}$, violates ϕ since there exists a run where a $send$ is not followed by a $success$. This behavior can be generated by replacing to the boxes $send_1$ and $send_2$ with a component that allow runs where a message is $sent$ and no $success$ is obtained, and an empty component that neither tries to $send$ the message again nor waits for a $success$, respectively.

Theorem 5.1.2 (Incomplete Model Checking complexity)**.** *Checking an IBA $\mathcal{M}$ against the BA $\overline{\mathcal{A}_\phi}$ associated to the LTL formula $\neg\phi$ has a temporal complexity $\mathcal{O}(|\mathcal{M}| \cdot |\overline{\mathcal{A}_\phi}|)$, where $|\mathcal{M}|$ and $|\overline{\mathcal{A}_\phi}|$ are the sizes of the model and the automaton associated with the negation of the claim, respectively.*

Proof. Checking an IBA $\mathcal{M}$ against a property expressed as a BA $\overline{\mathcal{A}_\phi}$ requires to check the emptiness of two intersection automata $\mathcal{I}_c$ and $\mathcal{I}$ representing a lower bound and an upper bound on the behaviors of the system, respectively. These automata contain in the words case $3 \cdot |Q_\mathcal{M}| \cdot |Q_{\overline{\mathcal{A}_\phi}}|$ states. The emptiness checking procedure is linear in the size of the considered automaton and has a $\mathcal{O}(|Q_\mathcal{I}| + |\Delta_\mathcal{I}|)$ temporal complexity. □

5.2 Constraint computation

When a property ϕ is possibly satisfied, each word v, recognized by the intersection automaton $\mathcal{I} = \mathcal{M} \cap \overline{\mathcal{A}_\phi}$, corresponds to a behavior $\mathcal{B}$ the system *may* exhibit that violates ϕ. To make ϕ satisfied, the developer must design the replacements $\mathcal{R}_{b_1}, \mathcal{R}_{b_2} \ldots \mathcal{R}_{b_n}$ of the black box states $b_1, b_2, \ldots b_n$ to forbid $\mathcal{B}$ from occurring. A replacement of a box is an automaton to be substituted to a box b. The goal of the constraint computation is to find the set of sub-properties, one for each box, to be satisfied by the replacements of the boxes of the model $\mathcal{M}$ such that the refinement $\mathcal{N}$ does not violate ϕ. Sub-properties are guidelines that help the developer in the replacement design and can be considered as a contract in a contract based design setting [88]. The constraint computation procedure is based on three subsequent steps: *a*) intersection cleaning; *b*) sub-properties generation; *c*) constraint identification.

5.2.1 Intersection cleaning

The intersection cleaning phase removes from the intersection automaton $\mathcal{I} = \mathcal{M} \cap \overline{\mathcal{A}_\phi}$ the states that are not involved in any behavior $\mathcal{B}$ that possibly violates the property. Indeed, these behaviors must not be included in any sub-property. Given the intersection automaton $\mathcal{I} = \mathcal{M} \cap \overline{\mathcal{A}_\phi}$, the cleaned intersection automaton $\mathcal{I}_\Upsilon$ is a version of the intersection automaton where the states from which it is not possible to reach an accepting state that can be entered infinitely many often are removed.

Definition 5.2.1 (Intersection cleaning)**.** *Given the intersection automaton $\mathcal{I} = \langle \Sigma_\mathcal{I}, Q_\mathcal{I}, \Delta_\mathcal{I}, Q^0_\mathcal{I}, F_\mathcal{I} \rangle$, the cleaned intersection automaton $\mathcal{I}_\Upsilon = \langle \Sigma_{\mathcal{I}_\Upsilon}, Q_{\mathcal{I}_\Upsilon}, \Delta_{\mathcal{I}_\Upsilon}, Q^0_{\mathcal{I}_\Upsilon}, F_{\mathcal{I}_\Upsilon} \rangle$ satisfies the following conditions:*

- $\Sigma_{\mathcal{I}_\Upsilon} = \Sigma_\mathcal{I}$;
- $Q_{\mathcal{I}_\Upsilon} = \{q \in Q_\mathcal{I}$ *such that there exits a possibly accepting run* ρ^ω *and an index* $i \geq 0$ *and* $\rho^\omega(i) = q\}$;
- $\Delta_{\mathcal{I}_\Upsilon} = \{(q, a, q') \in \Delta_\mathcal{I}$ *such that* $q \in Q_{\mathcal{I}_\Upsilon}$ *and* $q' \in Q_{\mathcal{I}_\Upsilon}\}$;
- $Q^0_{\mathcal{I}_\Upsilon} = Q^0_\mathcal{I} \cap Q_{\mathcal{I}_\Upsilon}$;
- $F_{\mathcal{I}_\Upsilon} = F_\mathcal{I} \cap Q_{\mathcal{I}_\Upsilon}$.

The cleaned version of the intersection automaton can be obtained using Algorithm 2. The algorithm first creates a copy of the intersection automaton $\mathcal{I}$ which will contain $\mathcal{I}_\Upsilon$ (Line 2). Then, the non trivial strongly connected components are computed and stored in the SCC set (Line 3). Then, the set Nxt is defined (Line 4). The set next is used to store the set of states from which it is possible to reach an accepting state (which can be entered infinitely often) whose predecessors still have to be analyzed.

Algorithm 2 Removes the states that are not involved in a possibly accepting run.

```
1:  procedure IntersectionCleaner(I)
2:      I_Υ ←Clone(I);
3:      SCC ←getNonTrivialSCC(I_Υ);
4:      Nxt ← {};
5:      for scc ∈ SCC do
6:          if scc ∩ F_I ≠ ∅ then
7:              Nxt ← Nxt ∪ scc;
8:          end if
9:      end for
10:     Vis ← {};
11:     while Nxt ≠ ∅ do
12:         s ←choose(Nxt);
13:         Vis ← Vis ∪ {s};
14:         Nxt ← Nxt \ {s};
15:         Nxt ← Nxt ∪ {s' | (s', a, s) ∈ Δ_I ∧ s ∉ Vis};
16:     end while
17:     for s ∈ (Q_I \ Vis) do
18:         removeState(I_Υ, s);
19:     end for
20: end procedure
```

For each set of states that form a strongly connected component scc which is in the SCC set (Line 5), if at least an accepting state is present (Line 6), the states are added to the set Nxt of the states to be visited next (Line 7). Then, the set Vis is defined (Line 10). The set Vis is used to store the set of the already visited states. The goal of this set is to guarantee that a state is not visited twice during the state space exploration. By exploring the state space of $\mathcal{I}$ the states from which it is possible to reach a state in the set Nxt are identified.

The state space exploration is an iterative process that ends when the set Nxt is empty (Line 11). At each exploration step, a state $s \in Nxt$ is selected (Line 12), added to the set of visited states (Line 13), and removed to the set Nxt (Line 14). Then, all the states s', which have not been already visited and are predecessors of the state s, are added to the set Nxt of the states to be analyzed next (Line 15). Finally, each state that has not been visited (Line 17) and its incoming and outgoing transitions are removed from the automaton (Line 18).

The intersection cleaning algorithm applied to the intersection automaton described in Figure 5.2 removes the states ④, ⑤ and ⑦ since they are not involved in any possibly accepting run.

Lemma 5.2.1 (Correctness)**.** *The cleaning procedure is correct.*

Proof. To prove that the procedure is correct, it is necessary to prove that the automaton $\mathcal{I}_\Upsilon$ obtained using Algorithm 2 satisfies the conditions specified in Definition 5.2.1.

Note that, the intersection cleaning algorithm is executed on the intersection automaton only when the property is possibly satisfied, i.e., it does not exists any accepting run, but the intersection only contains possibly accepting ones. The proof is by contradiction. Assume that there exists an automaton $\mathcal{I}_\Upsilon$ obtained using Algorithm 2 which does not satisfy Definition 5.2.1. Then, it must exist a state $q \in Q_{\mathcal{I}_\Upsilon}$ which is not involved in any possibly accepting run ρ^ω. Imagine that such a state exists. To not be removed at the end of the cleaning algorithm it must be contained into the set Vis of the visited states. To be inserted in Vis it is necessary that q was before inside the set Nxt. Two cases are possible: *a*) the state was included in the set Nxt before the `while` cycle. In this case, the state was a part of a strongly connected component that contains at least an accepting state. Thus, the hypothesis is contradicted, since from that state there was at least a possible way to reach an accepting state that is entered infinitely often; *b*) the state is included in the set Vis inside the `while` cycle. Then, the state is a predecessor of a state q' from which it is possible to reach an accepting state that can be entered infinitely often. Thus, the hypothesis is again contradicted. □

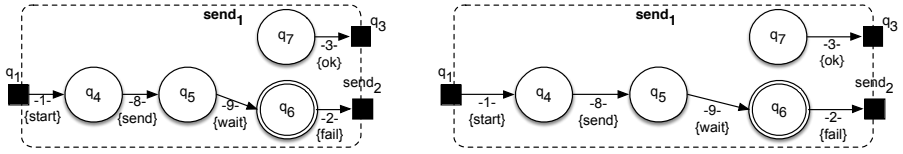
Theorem 5.2.1 (Complexity)**.** *The intersection cleaning procedure can be performed in time* $\mathcal{O}(|Q_\mathcal{I}| + |\Delta_\mathcal{I}|)$.

Proof. Cloning the intersection automaton $\mathcal{I}$ (Line 2) can be be done in time $\mathcal{O}(|Q_\mathcal{I}| + |\Delta_\mathcal{I}|)$. Indeed, it is sufficient to traverse $\mathcal{I}$ and clone its states and transitions. The same temporal complexity is required for finding the no trivial strongly connected components (Line 3). For example, it is possible to use the well known Tarjan's Algorithm [128]. Checking whether a strongly connected component contains an accepting state (Lines 4-9) can be done in time $\mathcal{O}(|Q_\mathcal{I}|)$. Finally, the state space exploration (Lines 11-16) has $\mathcal{O}(|Q_\mathcal{I}| + |\Delta_\mathcal{I}|)$ temporal complexity, since each state and transition is visited at most once. The same temporal complexity is required to remove the not reachable states (Lines 17-19). Thus, the final complexity of the algorithm is $\mathcal{O}(|Q_\mathcal{I}| + |\Delta_\mathcal{I}|)$. □

5.2.2 Sub-properties generation

The sub-property generation algorithm identifies for each box the corresponding sub-property. Each sub-property $\overline{\mathcal{S}_b}$ encodes the set of behaviors the replacement of the box b cannot exhibit. Indeed, these behaviors would lead to a *violation* or a *possible violation* of the properties of interest. For example, assume that the box $send_1$ presented in Figure 5.1a is accepting. If it is replaced by the automaton presented in Figure 5.3a, the claim $G(send \rightarrow F(success))$ is violated, since the sending activity is not followed by a success. Instead, if $send_1$ is replaced by the automaton presented in Figure 5.3b, it possibly violates the claim since it needs the "cooperation" of the replacement of the box $send_2$ to generate a violating behavior. Indeed, checking the refinement model after the first replacement is proposed yields to a claim violation, while in the second case the claim is possibly satisfied.

To encode the behaviors of the replacement of the box b that would lead to a violation of the claim ϕ, it is necessary to describe the internal violating behaviors and how these behaviors garnish the model of the system. For this reason, a sub-property $\overline{\mathcal{S}_b}$ associated with the box b provides *a*) a BA $\overline{\mathcal{P}_b}$, which describes the internal violating or possibly violating behaviors of the replacement, *b*) the incoming $\Delta^{in\overline{\mathcal{S}_b}}$ and outgoing $\Delta^{out\overline{\mathcal{S}_b}}$



(a) *A replacement for the box* $send_1$. **(b)** *A replacement for the box* $send_1$.

Figure 5.3: *Two examples of replacements for the box* $send_1$.

transitions, which describe how the replacement must (must not) be connected to the original model, *c*) function Π which describes the potential influence of the presence of these transitions on the claim satisfaction, *d*) the two reachability relations $\aleph$ and $\aleph_c$ which are used to encode how different runs of the automaton $\overline{\mathcal{P}_b}$ cooperate in the generation of a violation of ϕ. Formally:

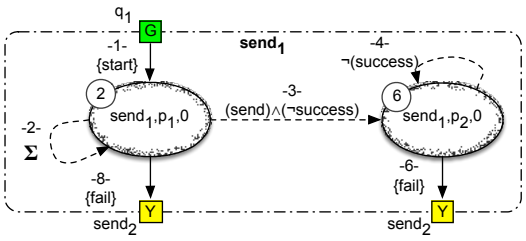
Definition 5.2.2 (Sub-property)**.** *Given a model* $\mathcal{M}$ *defined over the set of states* $Q_{\mathcal{M}}$*, a sub-property* $\overline{\mathcal{S}_b}$ *associated with the box* $b \in B_{\mathcal{M}}$*, corresponds to a tuple* $\overline{\mathcal{S}_b} = \langle \overline{\mathcal{P}_b}, \Delta^{in\overline{\mathcal{S}_b}}, \Delta^{out\overline{\mathcal{S}_b}}, \Pi, \aleph, \aleph_c, \Gamma^{\kappa}_{\mathcal{M}}, \Gamma^{\kappa}_{\overline{\mathcal{A}_\phi}} \rangle$*, where:*

- $\overline{\mathcal{P}_b} = \langle \Sigma_{\overline{\mathcal{P}_b}}, Q_{\overline{\mathcal{P}_b}}, \Delta_{\overline{\mathcal{P}_b}}, Q^0_{\overline{\mathcal{P}_b}}, F_{\overline{\mathcal{P}_b}} \rangle$ *is a BA.* $\overline{\mathcal{P}_b}$ *must satisfy the following conditions: if* $b \notin Q^0_{\overline{\mathcal{P}_b}}$ *then* $Q^0_{\overline{\mathcal{P}_b}} = \emptyset$*, if* $b \notin F_{\overline{\mathcal{P}_b}}$ *then* $F_{\overline{\mathcal{P}_b}} = \emptyset$*;*
- *the sets* $\Delta^{in\overline{\mathcal{S}_b}} \subseteq \{(q', a, q) \mid (q', a, b) \in \Delta_{\mathcal{M}}$ *and* $q \in Q_{\overline{\mathcal{P}_b}}$ *and* $q' \in Q_{\mathcal{M}}\}$ *and* $\Delta^{out\overline{\mathcal{S}_b}} \subseteq \{(q, a, q') \mid (b, a, q') \in \Delta_{\mathcal{M}}$ *and* $q \in Q_{\overline{\mathcal{P}_b}}$ *and* $q' \in Q_{\mathcal{M}}\}$ *are the incoming and outgoing transitions of the sub-property* $\overline{\mathcal{S}_s}$*;*
- $\Pi : \zeta \to \{G, Y, R\}$ *is a partial function over* $\zeta = \Delta^{in\overline{\mathcal{S}_b}} \cup \Delta^{out\overline{\mathcal{S}_b}}$*, which associates to the incoming and outgoing transitions a value in the set;*
- $\aleph \subseteq \Delta^{out\overline{\mathcal{S}_b}} \times \Delta^{in\overline{\mathcal{S}_b}}$ *and* $\aleph_c \subseteq \Delta^{out\overline{\mathcal{S}_b}} \times \Delta^{in\overline{\mathcal{S}_b}}$ *are two reachability relations;*
- $\Gamma^{\kappa}_{\mathcal{M}} : \kappa \to \{T, F\}$ *and* $\Gamma^{\kappa}_{\overline{\mathcal{A}_\phi}} : \kappa \to \{T, F\}$ *are the accepting functions, such that* $\kappa \in \{\aleph, \aleph_c\}$.

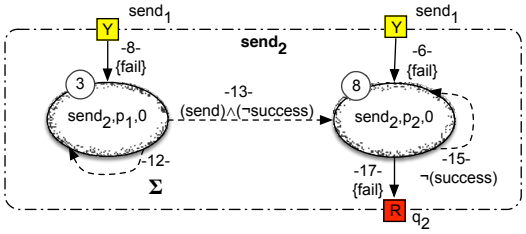
$\overline{\mathcal{P}_b}$ is the BA that encodes the condition the developer must satisfy in the design of the replacement $\mathcal{R}_b$ to be substituted to the black box state b. If the box b is not initial/accepting, the automaton $\overline{\mathcal{P}_b}$ can not contain initial/accepting states. The incoming $\Delta^{in\overline{\mathcal{S}_b}}$ and outgoing $\Delta^{out\overline{\mathcal{S}_b}}$ transitions specify how the behaviors encoded in the automaton $\overline{\mathcal{P}_s}$ are related to the original model $\mathcal{M}$. The function Π marks each of these transitions with a value in the set $\{G, Y, R\}$. An incoming transition $\delta \in \Delta^{in\overline{\mathcal{S}_b}}$ can be associated with the value G or Y, which specifies whether it is necessary to "traverse" at least a state of the intersection automaton generated from another box (Y) of the model, or all the runs that reach the incoming transition only involve states of the intersection automaton obtained from regular states of the model (G). An outgoing transition $\delta \in \Delta^{out\overline{\mathcal{S}_b}}$ can be associated with the value R or Y which specifies whether there is an accepting run (R) or a possibly accepting run (Y) starting from the destination state of the outgoing transition. The reachability relations $\aleph$ and $\aleph_c$ describe

how it is possible to reach an incoming transition from an outgoing transition of the sub-property in the intersection automaton. A tuple (δ^o, δ^i) is in $\aleph$ if and only if from the outgoing transition δ^o it is possible to reach, in the intersection automaton, the incoming transition δ^i without traversing a state of the intersection automaton generated from any other box of the model. Conversely, a tuple (δ^o, δ^i) is in $\aleph_p$ if and only if from the outgoing transition δ^o it is possible to reach incoming transition δ^i traversing at least another box different from b. $\Gamma^{\kappa}_{\mathcal{M}}$ and $\Gamma^{\kappa}_{\overline{\mathcal{A}_\phi}}$ associate to each reachability entry (δ_o, δ_i) in $\aleph$ and $\aleph_c$ a T or a F value. κ is assigned to $\aleph$ or $\aleph_c$ depending on the set of reachability entries considered[2]. Π, $\aleph$, $\aleph_c$, $\Gamma^{\kappa}_{\mathcal{M}}$ and $\Gamma^{\kappa}_{\overline{\mathcal{A}_\phi}}$ are described more in detail in the following of this thesis.

The sub-properties $\overline{\mathcal{S}_{send_1}}$ and $\overline{\mathcal{S}_{send_2}}$ associated with the boxes $send_1$ and $send_2$ of the model $\mathcal{M}$ described in Figure 5.1a and the claim ϕ, associated with the automaton $\overline{\mathcal{A}_\phi}$ described in Figure 5.1b, are presented in Figures 5.4a and 5.4b. The sub-properties are surrounded by a dotted border, which contains the name of the box the sub-property refers to. The automaton $\overline{\mathcal{P}_{send_1}}$, which corresponds to the sub-property $\overline{\mathcal{S}_{send_1}}$, contains the two states ② and ⑥ and the internal transitions $2, 3$ and 4. The sub-property $\overline{\mathcal{S}_{send_1}}$ is associated with the incoming transition 1 and the outgoing transitions 6 and 8. The



(a) *The sub-property $\overline{\mathcal{S}_{send_1}}$ that correspond to the box $send_1$.*



(b) *The sub-property $\overline{\mathcal{S}_{send_2}}$ that correspond to the box $send_2$.*

Figure 5.4: *The sub-properties associated with the boxes $send_1$ and $send_2$.*

automaton $\overline{\mathcal{P}_{send_2}}$, which corresponds to the sub-property $\overline{\mathcal{S}_{send_2}}$, contains the two states ③ and ⑧ and the internal transitions $12, 13$ and 15. The sub-property $\overline{\mathcal{S}_{send_2}}$ is associated with the incoming transitions 6, 8 and the outgoing transition 17. The function Π associates to the incoming transition 1 of the sub-property $\overline{\mathcal{S}_{send_1}}$ the value G since this transition is reachable from an initial state of the model through a run which does not involve the replacement associated with a box (graphically, the source/destination of an incoming/outgoing transition is depicted with a box which is decorated with the

[2] $\Gamma^{\kappa}_{\mathcal{M}}$ and $\Gamma^{\kappa}_{\overline{\mathcal{A}_\phi}}$ are a shortcut to represent the functions $\Gamma^{\aleph}_{\mathcal{M}}$, $\Gamma^{\aleph_c}_{\mathcal{M}}$ and $\Gamma^{\aleph}_{\overline{\Gamma_\phi}}$, $\Gamma^{\aleph_c}_{\overline{\mathcal{A}_\phi}}$, respectively.

name of the source/destination state and contains the value the function Π associates to the transition). Differently, Π marks the outgoing transitions of $\overline{\mathcal{S}_{send_1}}$ with the value Y since from these transitions it is possible to reach an accepting state through a run which involves the refinement of other boxes, i.e., of the box $send_2$. The outgoing transition 17 of the sub-property $\overline{\mathcal{S}_{send_2}}$ is associated with the value R since from the destination of the transition it is possible to reach an accepting state with a run that does not involve the refinement of other boxes. The incoming transitions 6 and 8 of the sub-property $\overline{\mathcal{S}_{send_2}}$ are associated with the value Y since the source of the transition is reachable through a run which involves the refinement of other boxes. Finally, the reachability relations $\aleph$ and $\aleph_c$ are empty since it is not possible to reach from an outgoing transition of the sub-property associated with the box $send_1$ ($send_2$) its incoming transition(s).

As the replacement, the sub-property $\overline{\mathcal{S}_b}$ is associated with the four types of accepting runs identified in Section 4.2.2: finite internal, infinite internal, finite external and infinite external accepting.

Given the cleaned intersection automaton $\mathcal{I}_\Upsilon$ (which contains the behaviors that possibly violate the claim), obtained from the intersection automaton $\mathcal{I} = \mathcal{M} \cap \overline{\mathcal{A}_\phi}$ between the model $\mathcal{M}$ and the automaton $\overline{\mathcal{A}_\phi}$, the sub-property identification problem concerns the identification of the sub-properties the developer must satisfy in the refinement activity. The sub-property identification procedure works through a set of subsequent steps: automata extraction, computation of the function Π and of the reachability relation.

First, the *automata extraction* procedure computes the automata associated with the sub-properties and their incoming and outgoing transitions.

Definition 5.2.3 (Automata extraction)**.** *Given a property ϕ which is possibly satisfied by the model $\mathcal{M}$, the cleaned intersection automaton $\mathcal{I}_\Upsilon$ obtained from the intersection automaton $\mathcal{I} = \mathcal{M} \cap \overline{\mathcal{A}_\phi}$, and the set $\zeta = \{\overline{\mathcal{S}_{b_1}}, \overline{\mathcal{S}_{b_2}}, \ldots \overline{\mathcal{S}_{b_n}}\}$ of the sub-properties associated to the black box states $b_1, b_2, \ldots b_n \in B_\mathcal{M}$ of $\mathcal{M}$ (at most one for each box), the automaton $\overline{\mathcal{P}_{b_i}}$ associated with each sub-property $\overline{\mathcal{S}_{b_i}} \in \zeta$ is a BA, such that:*

- $\Sigma_{\overline{\mathcal{P}_{b_i}}} = \Sigma_{\mathcal{I}_\Upsilon}$;
- $Q_{\overline{\mathcal{P}_{b_i}}} = \{\langle q_\mathcal{M}, p_{\overline{\mathcal{A}_\phi}}, x\rangle \in Q_{\mathcal{I}_\Upsilon}$ *such that* $q_\mathcal{M} = b_i\}$;
- $\Delta_{\overline{\mathcal{P}_{b_i}}} = \{(q, a, q')$ *such that* $q, q' \in Q_{\overline{\mathcal{P}}_{b_i}}$ *and* $(q, a, q') \in \Delta^p_\mathcal{I}\}$;
- $Q^0_{\overline{\mathcal{P}_{b_i}}} = Q^0_{\mathcal{I}_\Upsilon} \cap Q_{\overline{\mathcal{P}}_{b_i}}$;
- $F_{\overline{\mathcal{P}_{b_i}}} = F_{\mathcal{I}_\Upsilon} \cap Q_{\overline{\mathcal{P}}_{b_i}}$.

The incoming $\Delta^{in\overline{\mathcal{S}_{b_i}}}$ and outgoing $\Delta^{out\overline{\mathcal{S}_{b_i}}}$ transitions associated with the sub-property $\overline{\mathcal{S}_{b_i}}$ are defined as:

- $\Delta^{in\overline{\mathcal{S}_{b_i}}} = \{(q_\mathcal{M}, a, \langle b_i, p'_{\overline{\mathcal{A}_\phi}}, y\rangle)$ *such that* $(\langle q_\mathcal{M}, p_{\overline{\mathcal{A}_\phi}}, x\rangle, a, \langle b_i, p'_{\overline{\mathcal{A}_\phi}}, y\rangle) \in (\Delta^c_\mathcal{I} \cap \Delta_{\mathcal{I}_\Upsilon})\}$;
- $\Delta^{out\overline{\mathcal{S}_{b_i}}} = \{(\langle b_i, p_{\overline{\mathcal{A}_\phi}}, x\rangle, a, q'_\mathcal{M})$ *such that* $(\langle b_i, p_{\overline{\mathcal{A}_\phi}}, x\rangle, a, \langle q'_\mathcal{M}, p'_{\overline{\mathcal{A}_\phi}}, y\rangle) \in (\Delta^c_\mathcal{I} \cap \Delta_{\mathcal{I}_\Upsilon})\}$.

A sub-property $\overline{\mathcal{S}_{b_i}}$ associated to the box b_i is added to the set of sub-properties ζ if and only if $Q_{\overline{\mathcal{P}_{b_i}}} \neq \emptyset$.

The automata and the incoming and outgoing transitions of the sub-properties associated to the cleaned intersection automaton $\mathcal{I}_\Upsilon$ are computed using Algorithm 3. First, the algorithm considers the states $s_{\mathcal{I}_\Upsilon}$ of the intersection automaton (Line 2). If the corresponding state of the model q is a box (Line 3), the $s_{\mathcal{I}_\Upsilon}$ is added to the set $Q_{\overline{\mathcal{P}_q}}$ of the states of the sub-property $\overline{\mathcal{S}_q}$ (Line 4). If $s_{\mathcal{I}_\Upsilon}$ is initial (Line 5) or accepting (Line 8), it is also added to the set of initial (Line 6) and accepting (Line 9) states of $\overline{\mathcal{S}_q}$.

Algorithm 3 Identifies the automata associated with the sub-properties and their incoming and outgoing transitions.

1: **procedure** SUBPROPERTYIDENTIFICATION($\mathcal{I}_\Upsilon$, $\mathcal{M}$)
2: **for** $s_{\mathcal{I}_\Upsilon} \in Q_{\mathcal{I}_\Upsilon}$ **do**
3: **if** $s_{\mathcal{I}_\Upsilon} = \langle q, p, x\rangle \wedge q \in B_{\mathcal{M}}$ **then**
4: $Q_{\overline{\mathcal{P}_q}} \leftarrow Q_{\overline{\mathcal{P}_q}} \cup \{s_{\mathcal{I}_\Upsilon}\}$;
5: **if** $s_{\mathcal{I}_\Upsilon} \in Q^0_{\mathcal{I}_\Upsilon}$ **then**
6: $Q^0_{\overline{\mathcal{P}_q}} \leftarrow Q^0_{\overline{\mathcal{P}_q}} \cup \{s_{\mathcal{I}_\Upsilon}\}$;
7: **end if**
8: **if** $s_{\mathcal{I}_\Upsilon} \in F_{\mathcal{I}_\Upsilon}$ **then**
9: $F_{\overline{\mathcal{P}_q}} \leftarrow F_{\overline{\mathcal{P}_q}} \cup \{s_{\mathcal{I}_\Upsilon}\}$;
10: **end if**
11: **end if**
12: **end for**
13: **for** $(\langle b, p, x\rangle, a, \langle b, p', y\rangle) \in \Delta^p_{\mathcal{I}_\Upsilon}$ and $b \in B_{\mathcal{M}}$ **do**
14: $\Delta_{\overline{\mathcal{P}_b}} = \Delta_{\overline{\mathcal{P}_b}} \cup (\langle b, p, x\rangle, a, \langle b, p', y\rangle)$;
15: **end for**
16: **for** $(\langle q, p, x\rangle, a, \langle q', p', y\rangle) \in \Delta^c_{\mathcal{I}_\Upsilon}$ **do**
17: **if** $q \in B_{\mathcal{M}}$ **then**
18: $\Delta^{in\overline{\mathcal{S}_q}} \leftarrow \Delta^{in\overline{\mathcal{S}_q}} \cup (\langle q, p, x\rangle, a, \langle q', p', y\rangle)$;
19: **end if**
20: **if** $q' \in B_{\mathcal{M}}$ **then**
21: $\Delta^{out\overline{\mathcal{S}_q}} \leftarrow \Delta^{out\overline{\mathcal{S}_q}} \cup (\langle q, p, x\rangle, a, \langle q', p', y\rangle)$;
22: **end if**
23: **end for**
24: **end procedure**

Then, each transition $(\langle b, p, x\rangle, a, \langle b, p', y\rangle)$ of the intersection automaton which is possibly executed inside the box b is considered (Line 13) and added to the corresponding automaton (Line 14). Finally, each transition $(\langle q, p, x\rangle, a, \langle q', p', y\rangle)$ which is obtained by combining transitions of the model and of the claim is analyzed (Line 16). If the state of the model associated with the source state (Line 17) or the destination state (Line 20) of the transition is a box, the transition is added to the set of the outgoing (Line 18) or incoming (Line 21) transitions of the sub-property.

Theorem 5.2.2 (Automaton extraction complexity)**.** *The automaton extraction process can be performed in time $\mathcal{O}(|Q_{\mathcal{I}_\Upsilon}| + |\Delta_{\mathcal{I}_\Upsilon}|)$.*

Proof. Lines 2-12 of Algorithm 3 visit each state of the intersection automaton at most once, while Lines 13-23 visit each transition of the intersection automaton exactly once. In both the cases, at each step, a constant number of operations is executed inducing a $\mathcal{O}(|Q_{\mathcal{I}_\Upsilon}| + |\Delta_{\mathcal{I}_\Upsilon}|)$ temporal complexity . □

It is important to notice that every word v that is in the finite and infinite abstraction of the intersection automaton is a word associated with a sub-property and vice versa.

Theorem 5.2.3 (The language of the sub-property corresponds to the abstraction of the intersection automaton)**.** *Given the model $\mathcal{M}$ which possibly satisfies the claim ϕ, the intersection automaton $\mathcal{I} = \mathcal{M} \cap \mathcal{A}_\phi$ and the set of sub-properties ζ obtained as specified in Definition 5.2.3, for every box b:*

1. $v \in \alpha_b^*(\mathcal{I}) \Leftrightarrow v \in (\mathcal{L}^{e*}(\overline{\mathcal{S}_b}) \cup \mathcal{L}^{i*}(\overline{\mathcal{S}_b}))$;
2. $v \in \alpha_b^\omega(\mathcal{I}) \Leftrightarrow v \in (\mathcal{L}^{e\omega}(\overline{\mathcal{S}_b}) \cup \mathcal{L}^{i\omega}(\overline{\mathcal{S}_b}))$.

Theorem 5.2.3 specifies that the words of the finite abstractions of the intersection automaton associated with the box b correspond to the union of the external and internal finite words associated with the sub-property $\overline{\mathcal{S}_b}$ (condition 1). Furthermore, the words of the infinite abstractions of the intersection automaton associated with the box b correspond to the union of the external and internal infinite words associated with the sub-property $\overline{\mathcal{S}_b}$ (condition 2).

Proof. Let us first consider the statement 1.

($\Leftarrow$) Let us first consider the case in which v is in the set of finite externally accepted words associated with the sub-property $\overline{\mathcal{S}_b}$, i.e., $v \in \mathcal{L}^{e*}(\overline{\mathcal{S}_b})$. We want to construct a run $\rho_\mathcal{I}(i)$ whose abstraction corresponds to the word v in the intersection automaton. If such run exists we can conclude that $v \in \alpha_b^*(\mathcal{I})$. Thus, we write the word v as $\nu_{init}\nu^*\nu_{out}$. By definition, ν_{init} must be associated to an incoming transition of the sub-property. Thus, it is possible to associate to $\rho_\mathcal{I}(0)$ and $\rho_\mathcal{I}(1)$ the intersection states associated to the source and the destination of the incoming transition of the sub-property. Then, each state $\rho_{\overline{\mathcal{S}_b}}(i)$ of the run that makes the word ν^* externally accepted is associated with the corresponding state $\rho_\mathcal{I}(i)$ of the intersection automaton (which must exist from construction). This implies that the run $\rho_\mathcal{I}(1)$ satisfies the conditions 1 and 2 of Definition 5.1.2. Since $(\rho_{\overline{\mathcal{S}_b}}(i), v_i, \rho_{\overline{\mathcal{S}_b}}(i+1)) \in \Delta_{\overline{\mathcal{S}_b}}$ and $(\rho_\mathcal{I}(i), v_i, \rho_\mathcal{I}(i+1)) \in \Delta_\mathcal{I}^p$ by construction, it implies that the run $\rho_{\overline{\mathcal{S}_b}}$ is also executable on the automaton $\mathcal{I}$, i.e., condition 3 is satisfied. Furthermore, since by construction and Definition 4.2.6, $\rho_{\overline{\mathcal{S}_b}}(0) \in Q^{0in_b}$ (i.e., the source state of the run must be a source of an incoming transition) and $\rho_{\overline{\mathcal{S}_b}}(|v|) \in F^{out_b}$ (i.e., the last state of the run must be the destination of an outgoing transition), conditions 4 and 5 are satisfied. Since we have found a finite abstracted run $\rho_\mathcal{I}(i)$ associated to v and the box b, we conclude that $v \in \alpha_b^*(\mathcal{I})$.
The same approach applies to the case in which v is in the set of finite internally accepted words associated with the sub-property $\overline{\mathcal{S}_b}$, i.e., $v \in \mathcal{L}^{i*}(\overline{\mathcal{S}_b})$. The only difference concerns the initial state $\rho_{\overline{\mathcal{S}_b}}(0)$, which by Definition 4.2.6 must be an initial state of the sub-property, i.e., $\rho_{\overline{\mathcal{S}_b}}(0)$ is in $Q^0_{\overline{\mathcal{P}_b}}$ that by construction implies that $\rho_\mathcal{I}(0) \in Q^0_\mathcal{I}$. This implies that $\rho_{\overline{\mathcal{S}_b}}(0)$ can be considered as the initial state of the run corresponding to the word, making $i = 0$ in Definition 5.1.2 condition 5.

($\Rightarrow$) The proof is by contradiction assume that $v \in (\mathcal{L}^{e*}(\overline{\mathcal{S}_b}) \cup \mathcal{L}^{i*}(\overline{\mathcal{S}_b}))$ is false and $v^* \in \alpha_b^*(\mathcal{I})$ is true. Since $v \in \alpha_b^*(\mathcal{I})$, there exists an infinite possibly accepting run in $\mathcal{I}$ and v is an abstraction of the corresponding word which is associated to the box b. Since $\overline{\mathcal{S}_b}$ is obtained from the intersection automaton by aggregating the portions of the state space that refer to the box b, it follows that v must be associated with a run that

traverses $\overline{\mathcal{S}_b}$ (either starting from one of the initial states of $\overline{\mathcal{S}_b}$ and reaching one of its outgoing transitions, or starting from an incoming transition of $\overline{\mathcal{S}_b}$ and reaching one of its outgoing transitions). This implies that $v^* \in (\mathcal{L}^{e*}(\overline{\mathcal{S}_b}) \cup \mathcal{L}^{i*}(\overline{\mathcal{S}_b}))$ which makes the hypothesis contradicted.

The proof of the statement 2 corresponds to the proof 1 with the exception that it considers infinite words. □

The second step of the sub-property identification procedure is the *computation of the partial function* Π. Π associates to the incoming/outgoing transitions of the sub-property a value which specifies whether the initial/accepting state is reachable from the incoming/outgoing transition through a path that only contains only purely regular states, or it is necessary to traverse other mixed states i.e., the replacement of *other* boxes. The Π function is computed for each box $b \in B_{\mathcal{M}}$.

Definition 5.2.4 (Π function)**.** *Given a sub-property $\overline{\mathcal{S}_b}$, and given the set of its incoming and outgoing transitions $\zeta = \Delta^{in\overline{\mathcal{S}_b}} \cup \Delta^{out\overline{\mathcal{S}_b}}$ and the intersection automaton $\mathcal{I}$, Π is a partial function $\Pi : \zeta \rightarrow \{G, Y, R\}$, such that given a $\delta = (s, a, s')$ and $\delta \in \zeta$ the following are satisfied:*

1. $\Pi(\delta) = R \Leftrightarrow$ *exists $\rho^{\omega}_{\mathcal{I}}$ and an $i > 0$, such that $\rho^{\omega}(i) = s'$ and $(\rho^{\omega}(i-1), a, \rho^{\omega}(i)) = \delta$ and for all $j \geq i$, $\rho^{\omega}(j) \in PR_{\mathcal{I}}$;*
2. $\Pi(\delta) = G \Leftrightarrow$ *exists $\rho^{\omega}_{\mathcal{I}}$ and $i > 0$, such that $\rho^{\omega}(i) = s$ and $(\rho^{\omega}(i), a, \rho^{\omega}(i+1)) = \delta$ and for all $0 \leq j \leq i, \rho^{\omega}(j) \in PR_{\mathcal{I}}$;*
3. $\Pi(\delta) = Y \Leftrightarrow \Pi(\delta) \neq R$ *and* $\Pi(\delta) \neq G$ *and one of the following is satisfied:*
   - *exists $\rho^{\omega}_{\mathcal{I}}$ and $i > 0$, such that $\rho^{\omega}(i) = s'$ and $(\rho^{\omega}(i-1), a, \rho^{\omega}(i)) = \delta$ and for all $j \geq i, \rho^{\omega}(j) \notin Q_b$;*
   - *exists $\rho^{\omega}_{\mathcal{I}}$ and $i > 0$, such that $\rho^{\omega}(i) = s$ and $(\rho^{\omega}(i), a, \rho^{\omega}(i+1)) = \delta$ and for all $0 \leq j \leq i$, $\rho^{\omega}(j) \notin Q_b$).*

Definition 5.2.4 implies that incoming transitions can be marked as G and Y (or not be marked), while outgoing transitions can be marked as R and Y (or not be marked). This follows from the assumption that the constraint computation algorithm is performed in the cases in which the model possibly satisfies the claim ($\|\mathcal{M}^{\phi}\| = \bot$). This assumption implies the absence of an infinite run of purely regular states that connects an initial state of the intersection automaton to an accepting state that can be entered infinitely often, since this would imply that $\|\mathcal{M}^{\phi}\| = F$. The incoming transitions marked as G are the ones which are reachable from the initial state of the intersection automaton $\mathcal{I}$ without passing through mixed states, i.e., the incoming transitions are the transitions whose reachability does not depend on the replacements of other boxes. The outgoing transitions marked as R are the transitions from which an accepting state that can be entered infinitely often of the intersection automaton $\mathcal{I}$ is reachable without passing through mixed states, i.e., the outgoing transitions are the transitions from which it is possible to reach a state that makes the sub-property violated independently on the replacements of the other boxes. Finally, the incoming and outgoing transitions marked as Y are the transitions such that the reachability of the initial and accepting states depends on the replacement of *other* boxes (which are different from the box b

it-self). Given a sub-property $\overline{\mathcal{S}_b}$, the function Π specifies whether the corresponding runs violate or possibly violate the property ϕ of interest. For example, when the developer designs the replacement of a box b associated to a sub-property $\overline{\mathcal{S}_b}$ he/she must not design a component that allows $\overline{\mathcal{S}_b}$ to reach a outgoing port marked as R from an incoming port marked as G. Indeed, in this case, it is providing the system a way to reach from an initial state an accepting state of the intersection automaton which can be entered infinitely often.

The function Π can be computed using the procedure described in Algorithm 4. The algorithm works in two steps which mark the incoming (Lines 2-5) and outgoing (Lines 6-7) transitions of the sub-property $\overline{\mathcal{S}_b}$. The incoming transitions marked as G (Line 3) and Y (Line 4) are computed through the function FORWARD$\Pi\Pi$IDENTIFIER invoked over two different sets of states. When the incoming transitions to be marked as G are searched (Line 3), only purely regular states $PR_{\mathcal{I}}$ of the automaton $\mathcal{I}_\Upsilon$ are traversed, while, when the transitions to be marked as Y are considered (Line 4), all the states of the intersection automaton $\mathcal{I}_\Upsilon$ which are not states $Q_{\overline{\mathcal{P}_b}}$ of the automaton $\overline{\mathcal{P}_b}$ associated with the sub-property $\overline{\mathcal{S}_b}$ can be explored. The outgoing transitions marked as Y and R are computed through the method BACKWARD$\Pi\Pi$IDENTIFIER invoked on a different set of parameters. When the outgoing transitions to be marked as R are searched (Line 6), only the purely regular states $PR_{\mathcal{I}_\Upsilon}$ of the automaton $\mathcal{I}_\Upsilon$ are traversed, while, when the interest is on finding the incoming transitions to be marked as Y (Line 7), all the states of the intersection automaton $\mathcal{I}_\Upsilon$ which are not states $Q_{\overline{\mathcal{P}_b}}$ of the automaton $\overline{\mathcal{P}_b}$ associated with the sub-property $\overline{\mathcal{S}_b}$ are considered.

Algorithm 4 Computation of the function Π.

```
1: procedure ΠIDENTIFIER(I_Υ)
2:     for q_0 ∈ Q^0_I ∩ PR_{I_Υ} do
3:         FORWARDΠΠIDENTIFIER(q_0, I_Υ, PR_{I_Υ}, G);
4:         FORWARDΠΠIDENTIFIER(q_0, I_Υ, Q_{I_Υ} \ Q_{P̄_b}, Y);
5:     end for
6:     BACKWARDΠΠIDENTIFIER(I_Υ, PR_{I_Υ}, R);
7:     BACKWARDΠΠIDENTIFIER(I_Υ, Q_{I_Υ} \ Q_{P̄_b}, Y);
8: end procedure
```

The FORWARD$\Pi\Pi$IDENTIFIER procedure described in Algorithm 5 starts from a state s and searches for runs that involve only states in the set Q (passed as parameter) until an incoming transition of a sub-property $\overline{\mathcal{S}_b}$ is reached. Whenever a state s is visited by the algorithm, it is hashed (Line 2), then, each outgoing transition (s, a, s') of the state s (Line 3) is analyzed. If the destination state s' is in Q (Line 4) and it has not already been hashed (Line 5), then the FORWARD$\Pi\Pi$IDENTIFIER is continued (Line 6). If this is not the case, it means that the state s' is the destination of the incoming port (s, a, s'), thus, (s, a, s') is marked as C (where C is the value between G and Y passed as parameter) through the function Π (Line 9). Note that, an incoming transition is not marked as Y if it has been already marked as G.

Theorem 5.2.4 (FORWARD$\Pi\Pi$IDENTIFIER complexity)**.** *The procedure described in Algorithm 5 can be performed in time* $\mathcal{O}(|Q_{\mathcal{I}_\Upsilon}| + |\Delta_{\mathcal{I}_\Upsilon}|)$.

Proof. It is easy to prove that each state and transition of the cleaned intersection automaton $\mathcal{I}_\Upsilon$ is visited at most once, since it is visited if and only if it has not been

Algorithm 5 The procedure to find the incoming transitions to be marked as G and Y.

```
1:  procedure FORWARDΠIDENTIFIER(s, I_Υ, Q, C)
2:      hash(s);
3:      for (s, a, s') ∈ Δ_{I_Υ} do ;
4:          if s' ∈ Q then
5:              if s' not hashed then
6:                  FORWARDΠIDENTIFIER(s', I_Υ, Q, C);
7:              end if
8:          else
9:              Π(s, a, s') = C;
10:         end if
11:     end for
12: end procedure
```

hashed before. At each step, a finite and constant number of operations is performed leading the $\mathcal{O}(|Q_{\mathcal{I}_\Upsilon}| + |\Delta_{\mathcal{I}_\Upsilon}|)$ temporal complexity. □

The BACKWARDΠIDENTIFIER procedure is described in algorithm 6. The algorithm first looks for the non trivial strongly connected components that involve only states which are in the set Q (passed as parameter). To this purpose the algorithm constructs a version $\mathcal{I}_Q$ of $\mathcal{I}_\Upsilon$ that contains only the states of $\mathcal{I}_\Upsilon$ that belongs to Q (Line 2). Then, the non trivial strongly connected components of $\mathcal{I}_Q$ are identified (Line 3). The set $next$ is initialized to contain all the strongly connected components that contains at least a state which is accepting (Lines 4-9).

Then, the state space of $\mathcal{I}$ is explored to compute the outgoing transitions from which it is possible to reach one of the states in the set $next$. The set $visited$ (Line 10) is used to keep track of the already visited states of $\mathcal{I}$. The algorithm iteratively chooses a state s in the set $next$ (Lines 11,12), which is removed from $next$ (Line 14) and added to the set of visited states (Line 13). For each incoming transition $(s', a, s) \in \Delta_\mathcal{I}$ of s (Line 15), if the state s is a state of the sub-property (Line 16) the corresponding transition is marked with the value C passed as parameter (Line 17). Otherwise, if the purely regular state $'$ has not already been visited (Line 19) it is added to the set $next$ of states to be considered next (Line 20).

Algorithm 6 The procedure to find the outgoing transitions to be marked as R and Y.

```
1:  procedure BACKWARDΠIDENTIFIER(I_Υ, Q, C)
2:      I_Q ←ABSTRACT(I_Υ, Q);
3:      SCC ←GETNONTRIVIALSCC(I_Q);
4:      next ← {};
5:      for scc ∈ SCC do
6:          if scc ∩ F_I ≠ ∅ then
7:              next ← next ∪ scc;
8:          end if
9:      end for
10:     visited ← {};
11:     while next ≠ ∅ do
12:         s ← CHOOSE(next);
13:         visited ← visited ∪ {s};
14:         next ← next \ {s};
```

```
15:         for (s', a, s) ∈ Δ_I do
16:             if s' ∈ Q_{S_b^-} then
17:                 Π(s', a, s) = C;
18:             else
19:                 if s' ∉ visited then
20:                     next ← next ∪ {s'};
21:                 end if
22:             end if
23:         end for
24:     end while
25: end procedure
```

Theorem 5.2.5 (BACKWARDΠIDENTIFIER complexity)**.** *The procedure described in Algorithm 6 can be performed in time* $\mathcal{O}(|Q_{\mathcal{I}_\Upsilon}| + |\Delta_{\mathcal{I}_\Upsilon}|)$.

Proof. The algorithm first extracts a version $\mathcal{I}_Q$ of the intersection automaton which contains only the states in Q (Line 2). This can be done by exploring the state space of the intersection automaton $\mathcal{I}_\Upsilon$ and isolating the portions of the state space of interest. Then (Line 3), the list of non trivial strongly connected components SCC is isolated. This can be performed in time $\mathcal{O}(|Q_{\mathcal{I}_\Upsilon}| + |\Delta_{\mathcal{I}_\Upsilon}|)$, for example by using the well known Tarjan's Algorithm [128]. The only states added to the set $next$ are the states that belong to a strongly connected component which contains at least one accepting state (Lines 5-8). Starting from these states, the backward search is performed (Lines 11-24). This search visits each state and transition at most once. Thus, the BACKWARDΠIDENTIFIER algorithm has a $\mathcal{O}(|Q_{\mathcal{I}_\Upsilon}|+|\Delta_{\mathcal{I}_\Upsilon}|)$ temporal complexity. □

The last step of the sub-property identification procedure concerns the *computation of the reachability relation*. The reachability relation specifies how the presence of a run that traverses a sub-property influences the reachability of another run that traverses the sub-property itself. Imagine for example that the high level model of the system is the one presented in Figure 5.5. Differently from the model presented in Figure 4.1, in this case, the box $send_2$ is not contained in the IBA and $send_1$ can be left also through a transition that moves the system into the state q_4. Whenever the system reaches the state q_4, a timer is started. The transition 7 is fired whenever the $timer\_ack$ proposition is true, i.e., the system is notified that the time has been elapsed. The developer can now choose to propose a replacement for the box $send_1$ that behaves as follows. Whenever the replacement is entered through the transition 1, a sending activity is performed. If the sending activity succeeds, the transition 3 is fired, otherwise, the transition 6 which activates the timer is performed. If, instead, the replacement is entered through the transition 7 and the $send$ activity fails, the transition 2 is fired. Imagine that one of the properties of the system specifies that only one sending message activity must be performed by the system. The developer must know that if he/she designs the system such that a $send$ activity is performed on a run that connects 1 to 6, he/she can not replicate this activity in the run that connects 7 to 2 and vice versa. This is exactly the purpose of the reachability relation which specifies how the internal runs of a sub-property influence each others.

The reachability relation is computed by *abstracting* the portion of the state space that connects the two runs of the sub-property in the intersection automaton, and spec-

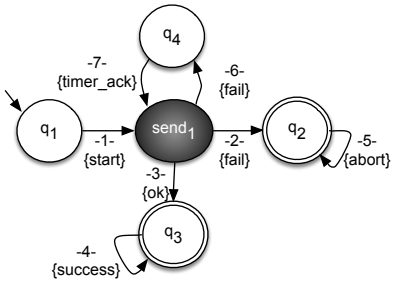


Figure 5.5: *A modeling alternative for the sending message protocol example.*

ifies if from an outgoing transition $\delta_{out} \in \Delta^{out\overline{\mathcal{S}_b}}$ of the sub-property it is possible to reach its incoming transition $\delta_{in} \in \Delta^{in\overline{\mathcal{S}_b}}$. Two versions of the reachability relation can be considered depending on whether the replacement of other boxes can be traversed, *upper reachability relation*, or not, *lower reachability relation*. In the first case, other boxes can be crossed, i.e., the replacements of other boxes allow to reach their outgoing transitions, while in the first case, only the behavior of the system which has been already specified is considered. The reachability relation will be graphically specified through directed dotted edges which connects the outgoing transitions of the sub-property with the corresponding incoming transitions.

Definition 5.2.5 (Lower reachability relation)**.** *Given a sub-property $\overline{\mathcal{S}_b}$ associated with the intersection automaton $\mathcal{I}$ and the set of its incoming and outgoing transitions $\zeta = \Delta^{in\overline{\mathcal{S}_b}} \cup \Delta^{out\overline{\mathcal{S}_b}}$, the lower reachability relation $\aleph_c = \Delta^{out\overline{\mathcal{S}_b}} \times \Delta^{in\overline{\mathcal{S}_b}}$ is a relation, such that given an outgoing transition $\delta_o = (s, a, s')$ and an incoming transition $\delta_i = (s'', a, s''')$, $(\delta_o, \delta_i) \in \aleph_c$ if and only if one of the following conditions is satisfied:*

1. *$\delta_o = \delta_i$;*
2. *there exist an accepting run ρ^ω, and two indexes $i, j \geq 0$ such that $s' = \rho^\omega(i)$ and $s'' = \rho^\omega(j)$ and for all k, such that $j \geq k \geq i$, $\rho^\omega(k) \in PR_{\mathcal{I}_\Upsilon}$.*

In other words, the reachability relation specifies how outgoing and incoming transitions of the sub-property are connected in the intersection automaton.

Proposition 5.2.1. *In the worst case, the lower reachability relation has a dimension which is $\mathcal{O}(|\Delta^{out\overline{\mathcal{S}_b}}| \cdot |\Delta^{in\overline{\mathcal{S}_b}}|)$.*

Differently from the lower reachability relation, in the upper reachability relation, an outgoing and an incoming transitions of the sub-property $\overline{\mathcal{S}_b}$-which refer to a black box state b of the model $\mathcal{M}$-are connected if and only if there exists a run between them that potentially involves also mixed states of the intersection automaton that do not refer to b, i.e., any state in $Q_{\mathcal{I}} \setminus Q_{\overline{\mathcal{P}_b}}$.

Definition 5.2.6 (Upper reachability relation)**.** *Given a sub-property $\overline{\mathcal{S}_b}$ associated to the intersection automaton $\mathcal{I}$ and the set of its incoming and outgoing transitions $\zeta = \Delta^{in\overline{\mathcal{S}_b}} \cup \Delta^{out\overline{\mathcal{S}_b}}$, the upper reachability relation $\aleph = \Delta^{out\overline{\mathcal{S}_b}} \times \Delta^{in\overline{\mathcal{S}_b}}$ is a relation such that, given an outgoing transition $\delta_o = (s, a, s')$ and an incoming transition $\delta_i = (s'', a, s''')$, $(\delta_o, \delta_i) \in \aleph$ if and only if one of the following is satisfied:*

1. $\delta_o = \delta_i$;
2. *there exist an accepting run* ρ^ω *and two indexes* $i, j \geq 0$, *such that* $s' = \rho^\omega(i)$ *and* $s'' = \rho^\omega(j)$ *and for all* k *such that* $j \geq k \geq i, \rho^\omega(k) \notin Q_{\overline{\mathcal{P}_b}}$.

Proposition 5.2.2. *In the worst case, the upper reachability relation has a dimension which is* $\mathcal{O}(|\Delta^{out\overline{\mathcal{S}_b}}| \cdot |\Delta^{in\overline{\mathcal{S}_b}}|)$.

The procedure used to compute the *upper* and the *lower* reachability relation is described in Algorithm 7, where $\Delta^{out\overline{\mathcal{S}_b}}$ and $\Delta^{in\overline{\mathcal{S}_b}}$ are the outgoing and incoming transitions of the sub-property $\overline{\mathcal{S}_b}$ to be considered, $\mathcal{I}_\Upsilon$ is the intersection automaton, Q is the set of the states to be considered in the computation of the reachability relation and $\sqsupseteq$ represents the lower ($\aleph_c$) or the upper ($\aleph$) reachability relation. When the upper reachability relation of a sub-property $\overline{\mathcal{S}_b}$ is computed, all the states of the intersection automaton with the exception of the states in $Q_{\overline{\mathcal{S}_b}}$ are considered; when the lower reachability relation is considered Q contains the purely regular states of the intersection automaton.

The procedure described in Algorithm 7 first computes an abstraction of the state space which only contains the states in the set Q (Line 2). Then (Line 3), for every pair of states (s, s'), the Floyd-Warshall algorithm [58] is used to compute if s' is reachable from s. Then, each incoming (s, a, s') (Line 4) and outgoing (s'', a, s''') (Line 5) tran-

Algorithm 7 The procedure to compute the reachability relation.

```
1:  procedure REACHABILITYRELATIONIDENTIFIER(Δ^{outS̄_b}, Δ^{inS̄_b}, I_Υ, Q, ⊒)
2:      I_Q ←ABSTRACT(I_Υ, Q);
3:      Rec ← FLOYDWARSHALL(I_Q);
4:      for (s, a, s') ∈ Δ^{outS̄_b} do
5:          for (s'', a, s''') ∈ Δ^{inS̄_b} do
6:              if ((s', s'') ∈ Rec) or ((s, a, s') = (s'', a, s''')) then
7:                  ⊒ = ⊒ ∪ ⟨(s, a, s'), (s'', a, s''')⟩;
8:              end if
9:          end for
10:     end for
11: end procedure
```

sition is analyzed. If it is possible to reach s'' from s' (Line 6), it means that there exists a run which contains only states in Q which allows to reach (s, a, s') from (s'', a, s'''). Thus, the pair $\langle (s, a, s'), (s'', a, s''') \rangle$ is added to the reachability relation $\sqsupseteq$ (Line 7).

Theorem 5.2.6 (REACHABILITYRELATIONIDENTIFIER correctness)**.** *The procedure described in Algorithm 7 is correct.*

Proof. Let us first consider the case in which the lower reachability relation is considered. It is necessary to prove that $(\delta_o, \delta_i) \in \aleph_c$ if and only if one of the conditions 1 or 2 of Definition 5.2.5 is satisfied.

($\Leftarrow$) If δ_o is equal to δ_i (condition 1), (δ_o, δ_i) is added in the reachability relation $\aleph_c$ in Line 7 since the condition in Line 6 is satisfied. If instead there exists a run, which contains only purely regular states, that connects the state s' to the state s'' (condition 2), the tuple (s', s'') is added to the relation Rec returned by the Floyd-Warshall algorithm,

which makes the condition in Line 6 satisfied and implies that (δ_o, δ_i) is added to the reachability relation $\aleph_c$.

($\Rightarrow$) If a tuple (δ_o, δ_i) belongs to $\aleph_c$, the procedure described in Algorithm 7 has added it to the lower reachability relation. To be added to this relation two cases are possible: *a*) the first clause of condition specified in Line 6 is triggered. Since (s', s'') is in Rec it must exists a run made by states contained in the set Q which allow to reach s'' from s' which implies that the condition 1 is satisfied; *b*) the second clause of the condition specified in Line 6 is satisfied. In this case δ_o is equal to δ_i which makes the condition 2 satisfied.

The same approach can be used to demonstrate that the procedure is correct when the upper reachability relation is considered. □

Theorem 5.2.7 (Constraint computation complexity)**.** *Given a sub-property $\overline{\mathcal{S}_b}$, associated with the box b, the procedure described in Algorithm 7 can be executed in $\mathcal{O}(|Q^3| + |\Delta^{out\overline{\mathcal{S}_b}}| \cdot |\Delta^{in\overline{\mathcal{S}_b}}|)$.*

Proof. As previously mentioned the abstraction procedure (Line 2) can be executed in time $\mathcal{O}(|Q_{\mathcal{I}_\Upsilon}| + |\Delta_{\mathcal{I}_\Upsilon}|)$. The Floyd Warshall algorithm (Line 3) has a temporal complexity $\mathcal{O}(Q^3)$, while the steps described in Lines 4-10 have a $|\Delta^{out\overline{\mathcal{S}_b}}| \cdot |\Delta^{in\overline{\mathcal{S}_b}}|$ complexity. Thus, the final complexity of the algorithm is $\mathcal{O}(|Q^3| + |\Delta^{out\overline{\mathcal{S}_b}}| \cdot |\Delta^{in\overline{\mathcal{S}_b}}|)$. □

Together with the reachability relation the functions $\Gamma^{\kappa}_{\mathcal{M}}$, $\Gamma^{\kappa}_{\overline{\mathcal{A}_\phi}}$ are computed, where κ may refer to $\aleph$ or $\aleph_c$. For each tuple $(\delta_i, \delta_o) \in \aleph$ and $(\delta_i, \delta_o) \in \aleph_c$ these functions specify whether there exists a run that connect the outgoing and the incoming transitions that contains an intersection state made by an accepting state of the model ($\Gamma^{\kappa}_{\mathcal{M}}$) and an intersection state made by an accepting state of the sub-property ($\Gamma^{\kappa}_{\overline{\mathcal{A}_\phi}}$), respectively. These functions specify the developer whether the presence of an accepting state in the replacement may lead to a violating run in the cases in which fairness conditions are considered.

A *constraint* Γ contains the set of sub-properties ζ on the replacements of the boxes $b_1, b_2, \ldots b_n \in B_{\mathcal{M}}$ that guarantee that ϕ is satisfied. Furthermore, for each box b the constraint contains a function $\mathcal{U}$ which specifies whether exists in the intersection automaton $\mathcal{I}_\Upsilon$ a possible violating run which does not involve any state of the intersection automaton generated by the box $b \in B_{\mathcal{M}}$.

Definition 5.2.7 (Constraint)**.** *Given the cleaned intersection automaton $\mathcal{I}_\Upsilon$ of the intersection automaton $\mathcal{I} = \mathcal{M} \cap \overline{\mathcal{A}_\phi}$ obtained from the IBA $\mathcal{M}$ and the BA $\overline{\mathcal{A}_\phi}$, the constraint $\mathcal{C}$ is a tuple $\langle \zeta, \mathcal{U} \rangle$, where ζ is obtained as specified in Definition 5.2.3 and $\mathcal{U} : B_{\mathcal{M}} \rightarrow \{T, F\}$, such that*

- *$\mathcal{U}(b) = T \Leftrightarrow$ there exists an accepting run ρ^ω in $\mathcal{I}_\Upsilon$ such that for all $i \geq 0, \rho^\omega(i) \notin Q_{\overline{\mathcal{P}}_b}$*

The value of the function $\mathcal{U}$ can be computed by running $|B_{\mathcal{M}}|$ times the emptiness checking procedure on the automaton $\mathcal{I}_\Upsilon$ (each time by removing the portion of the state space that refers to a box b under analysis).

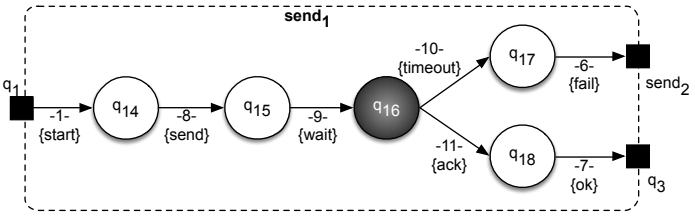
5.3 Replacement checking

At each refinement round $i \in \mathcal{RR}$, the developer produces a replacement $\mathcal{R}_b$ for a black box state b. The developer may want to check whether the properties of interest are satisfied by the new design. The refinement checking problem is the problem of checking whether the replacement makes the original property ϕ satisfied, not satisfied or possibly satisfied.

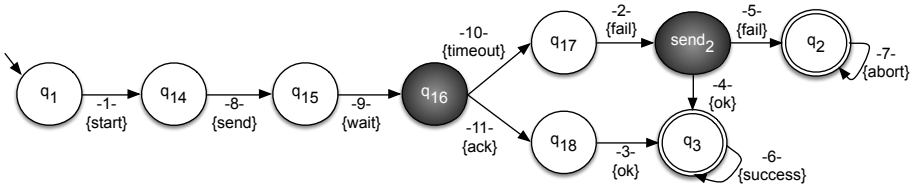
The replacement checking problem can be formulated as follows:

Definition 5.3.1 (Refinement Checking)**.** *Given a refinement round $i \in \mathcal{RR}$, where the developer refines the box b of $\mathcal{M}$ through the replacement $\mathcal{R}_b$, the refinement checking problem is to compute whether the refined automaton $\mathcal{N}$, obtained by composing the replacement $\mathcal{R}_b$ of the box b and the model $\mathcal{M}$, satisfies, does not satisfy or possibly satisfies the property ϕ.*

For example, assume that the box $send_1$ of the model $\mathcal{M}$, presented in Figure 4.1, is refined using the replacement $\mathcal{R}_{send_1}$ described in Figure 5.6a. The replacement $\mathcal{R}_{send_1}$, after it is entered through the incoming transition labeled with $start$, reaches the state q_{14}. Then, the message is sent and the system moves from q_{14} to q_{15}. After the sending activity, the system waits for a notification by moving to the state q_{16}. The state q_{16} is a box, meaning that it still has to be refined. When the replacement of q_{16} is left, two



(a) *The replacement $\mathcal{R}_{send_1}$ of the box $send_1$.*



(b) *The refinement $\mathcal{N}$ obtained by plugging the replacement $\mathcal{R}_{send_1}$ into the model $\mathcal{M}$.*

Figure 5.6: *The replacement $\mathcal{R}_{send_1}$ of the model $\mathcal{M}$ presented in Figure 4.1 and the refinement $\mathcal{N}$ obtained by plugging the replacement $\mathcal{R}_{send_1}$ into the model $\mathcal{M}$.*

cases are possible: a $timeout$ event occurs and the system moves to the state q_{17} or an ack message is received which makes the system moving to q_{18}. In the first case, the replacement is left through the transition labeled with $fail$ which moves the system to the state $send_2$. In the second, the replacement is left through the transition labeled with ok which moves the system to the state q_3. The replacement checking problem is the problem of verifying whether the refined model $\mathcal{N}$ (the initial model $\mathcal{M}$ plus the replacement of the box) satisfies the original property ϕ. The refined model $\mathcal{N}$ is described in Figure 5.6b.

Whenever a replacement for a black box state b is proposed the idea is to not consider all the model from scratch, by generating its refinement $\mathcal{N}$, but to only consider the replacement $\mathcal{R}_b$ against the previously generated constraint $\mathcal{C}$ and more precisely the corresponding sub-property $\overline{\mathcal{S}_b}$. For example, the replacement of the box $send_1$ can be considered in relation to the sub-property $\overline{\mathcal{S}_{send_1}}$ specified in Figure 5.4a. The sub-property specifies that any *finite* path that crosses the box $send_1$, entering the box by means of the incoming transition 1 (which arrives from the state q_1) and leaving the replacement through a transition marked with $fail$ (which reaches the state $send_2$) is a possibly violating run. The run is possibly violating since if we do not satisfy the constraint we cannot claim that the property is not satisfied, since the violation may depend on the replacement proposed for the other boxes. Similarly, the possibly violating runs also include any *finite* run entering from the transition which arrives from the state q_1 in which a $send$ is not followed by a $success$ before leaving the replacement through a transition marked with $fail$ and with destination the state $send_2$.

Checking whether a replacement satisfies a sub-property can be reduced to two emptiness checking problems. The first emptiness checking procedure considers an automaton which encodes the set of behaviors the system is going to exhibit at run-time (an *under* approximation), and checks whether the property ϕ is violated. The second analyzes an automaton which also contains the behaviors the system may exhibit (an *over* approximation). The under and the over approximation automaton are generated starting from a common intersection automaton. Section 5.3.1 describes the intersection between a replacement and the corresponding sub-property and how the under and the over approximation are obtained from this intersection. Section 5.3.2 presents the replacement checking procedure.

5.3.1 Intersection between a sub-property and replacement

The basic version of the intersection automaton between a replacement $\mathcal{R}_b$ and the sub-property $\overline{\mathcal{S}_b}$ of the box b, from which the under and over approximation are computed, is described in Definition 5.3.2.

Definition 5.3.2 (Intersection between a sub-property and a replacement)**.** *Given the sub-property* $\overline{\mathcal{S}_b} = \langle \overline{\mathcal{P}_b}, \Delta^{in\overline{\mathcal{S}_b}}, \Delta^{out\overline{\mathcal{S}_b}}, \Pi, \aleph, \aleph_c, \Gamma^{\kappa}_{\mathcal{M}}, \Gamma^{\kappa}_{\overline{\mathcal{A}_\phi}} \rangle$ *associated with the box* b*, and the replacement* $\mathcal{R}_b = \langle \mathcal{M}_b, \Delta^{inR_b}, \Delta^{outR_b} \rangle$*, the intersection* $\mathcal{I} = \overline{\mathcal{S}_b} \cap \mathcal{R}_b$ *is a tuple* $\langle \mathcal{I}_b, \Delta^{inI_b}, \Delta^{outI_b}, \Pi_b \rangle$ *such as:*

- $\mathcal{I}_b$ *is the intersection automaton.* $\mathcal{I}_b$ *is obtained as specified in Definition 5.1.1, considering* $\mathcal{M}_b$ *and* $\overline{\mathcal{P}_b}$ *as model and claim, respectively;*
- $\Delta^{inI_b} = \{(q, a, \langle q', p, x \rangle) \mid (q, a, q') \in \Delta^{inR_b}, (q, a, p) \in \Delta^{in\overline{\mathcal{S}_b}}$ *and* $x \in \{0, 1, 2\}\}$*;*
- $\Delta^{outI_b} = \{(\langle q', p, x \rangle, a, q) \mid (q', a, q) \in \Delta^{outR_b}, (p, a, q) \in \Delta^{out\overline{\mathcal{S}_b}}$ *and* $x \in \{0, 1, 2\}\}$*;*
- $\Pi_b : \Delta^{inI_b} \cup \Delta^{outI_b} \to \{G, Y, R\}$, $\Pi(\delta) = \Pi(\delta')$ *where* δ' *is the incoming/outgoing transition of* $\overline{\mathcal{S}_b}$ *from which* δ *is obtained.*

Informally, the intersection between a replacement $\mathcal{R}_b$ and the corresponding sub-property $\overline{\mathcal{S}_b}$ is an automaton which is obtained by the intersection of the automata

associated with $\mathcal{R}_b$ and $\overline{\mathcal{S}_b}$ and a set of incoming and outgoing transitions that corresponds to the synchronous execution of the transitions of $\mathcal{R}_b$ and $\overline{\mathcal{S}_b}$. For example, the intersection between the replacement $\mathcal{R}_b$ described in Figure 5.6a and the corresponding sub-property presented in Figure 5.4a is presented in Figure 5.7[3]. For each $\delta \in (\Delta^{inI_b} \cup \Delta^{outI_b})$ we say that $\Pi(\delta) = \Pi(\delta')$, where δ' is the incoming/outgoing transition of the sub-property from which δ is obtained.

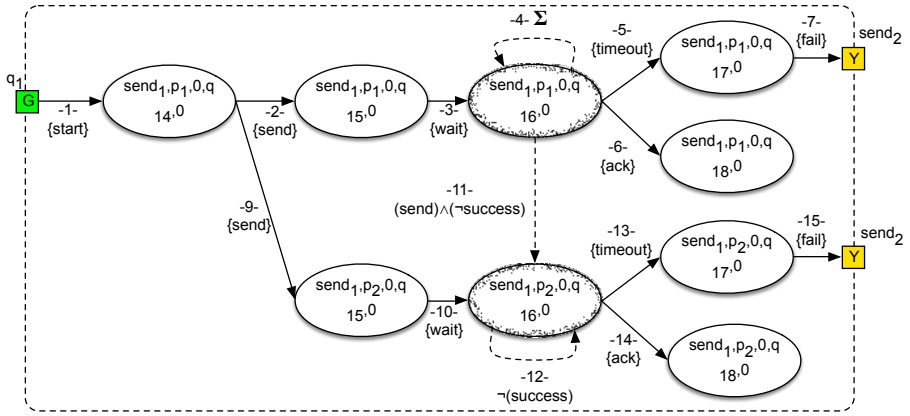


Figure 5.7: *Intersection between the replacement $\mathcal{R}_{send_1}$ described in Figure 5.6a and the sub-property $\overline{\mathcal{S}_{send_1}}$ presented in Figure 5.4a.*

As for the case of IBA, we define as $\mathcal{I}_c$ the intersection obtained considering the completeness version $\mathcal{M}_{b_c}$ of $\mathcal{M}_b$. The intersection between the sub-property $\overline{\mathcal{S}_b} = \langle \overline{\mathcal{P}_b}, \Delta^{in\overline{\mathcal{S}_b}}, \Delta^{out\overline{\mathcal{S}_b}}, \Pi, \aleph, \aleph_c, \Gamma^{\kappa}_{\mathcal{M}}, \Gamma^{\kappa}_{\overline{\mathcal{A}_\phi}} \rangle$ associated to the box b and the replacement $\mathcal{R}_b = \langle \mathcal{M}_b, \Delta^{inR_b}, \Delta^{outR_b} \rangle$ has the same structure of a replacement (Defined in 4.2.3), i.e., it contains an automata and a set of incoming and outgoing transitions, and it can be associated with finite internal, infinite internal, finite external and infinite external accepting runs as defined in Section 4.2.2.

Lemma 5.3.1 (Finite internal intersection language)**.** *The intersection automaton $\mathcal{I} = \mathcal{R}_b \cap \overline{\mathcal{S}_b}$ between the replacement $\mathcal{R}_b$ and the sub-property $\overline{\mathcal{S}_b}$ recognizes the finite internal language $\mathcal{L}^{i*}(\mathcal{I}) = (\mathcal{L}^{i*}(\mathcal{R}_b) \cup \mathcal{L}^{i*}_p(\mathcal{R}_b)) \cap \mathcal{L}^*(\overline{\mathcal{S}_b})$, i.e., $v \in \mathcal{L}^{i*}(\mathcal{I}) \Leftrightarrow v \in ((\mathcal{L}^{i*}(\mathcal{R}_b) \cup \mathcal{L}^{i*}_p(\mathcal{R}_b)) \cap \mathcal{L}^{i*}(\overline{\mathcal{S}_b}))$.*

Proof. ($\Rightarrow$) If a word $v \in \mathcal{L}^{i*}(\mathcal{I})$, it must exists a finite internal run ρ^* in the intersection automaton $\mathcal{I}$ where the initial states are the initial states of the intersection automaton and the final states are the destinations of the outgoing transitions of $\mathcal{I}$. Since $\rho(0)$ must be an initial state of the intersection automaton, it must be obtained from an initial state of the automaton $\mathcal{R}_b$ associated with the replacement of the black box b. Let us identify with $\rho_{\mathcal{R}_b}(0)$ the state of the replacement $\mathcal{R}_b$ from which $\rho(0)$ is obtained. For each i, such that $0 \leq i < |\rho|$, for each transition $(\rho(i), a, \rho(i+1))$ that moves the system from the state $\rho(i)$ to the state $\rho(i+1)$, it must exist a transition $(\rho_{\mathcal{R}_b}(i), a, \rho_{\mathcal{R}_b}(i+1))$ in the automaton $\mathcal{R}_b$, which corresponds to the replacement of the box b, or $\rho_{\mathcal{R}_b}(i) = \rho_{\mathcal{R}_b}(i+1)$ and $\rho_{\mathcal{R}_b} = b$[4]. Since the last state of the run $\rho(|v|)$ is the destination of an

[3] Note that Figure 5.7 only contains the portion of the state space where $x = 0$.
[4] This follows from the definition of the intersection (see Definition 5.1.1).

outgoing transition of $\mathcal{I}$, and the outgoing transitions of the intersection are obtained by synchronously executing transitions of the replacement and the sub-property, the transition must also be outgoing for the replacement. This implies that $\rho_{\mathcal{R}_b}(i)$ is a finite internal run (accepting or possibly accepting depending on the presence of boxes) for the replacement $\mathcal{R}_b$. The same reasoning can be applied to demonstrate that v is contained in the language $\mathcal{L}^*(\overline{\mathcal{S}_b})$.
($\Leftarrow$) The proof is by contradiction. Assume that there exists a word $v \notin \mathcal{L}^{i*}(\mathcal{I})$ which is in $((\mathcal{L}^{i*}(\mathcal{R}_b) \cup \mathcal{L}_p^{i*}(\mathcal{R}_b)) \cap \mathcal{L}^*(\overline{\mathcal{S}_b}))$. Since $v \in ((\mathcal{L}^{i*}(\mathcal{R}_b) \cup \mathcal{L}_p^{i*}(\mathcal{R}_b)) \cap \mathcal{L}^*(\overline{\mathcal{S}_b}))$, it must exist a finite run $\rho_{\mathcal{R}_b}$ in the replacement and in the sub-property $\rho_{\overline{\mathcal{S}_b}}$ associated with v. Given the initial states $\rho_{\mathcal{R}_b}(0)$ and $\rho_{\overline{\mathcal{S}_b}}(0)$ of the model and the claim, respectively, from which the initial state of the run is obtained, it must exist by construction a state s in the intersection automaton $\mathcal{I}$ which is obtained by combining these two states. This state by construction is also initial for the intersection automaton. Let us identify with $\rho_{\mathcal{I}}$ the run that starts from this state. For each $0 < i < |v|-1$, $\rho_{\mathcal{I}}(i+1)$ is associated to the state of the intersection automaton obtained by combining $\rho_{\mathcal{R}_b}(i+1)$ and $\rho_{\overline{\mathcal{S}_b}}(i+1)$. Note that if $\rho_{\mathcal{R}_b}(i)$ and $\rho_{\overline{\mathcal{S}_b}}(i)$ are connected to $\rho_{\mathcal{R}_b}(i+1)$ and $\rho_{\overline{\mathcal{S}_b}}(i+1)$ with a transition labeled with v_i, or $\rho_{\overline{\mathcal{S}_b}}(i)$ is connected to $\rho_{\overline{\mathcal{S}_b}}(i+1)$ and $\rho_{\mathcal{R}_b}(i)$ is a box, then $\rho_{\mathcal{I}}(i)$ and $\rho_{\mathcal{I}}(i+1)$ are connected by a transition labeled with v_i by construction. Finally, the states $\rho_{\mathcal{R}_b}(|v|)$ and $\rho_{\overline{\mathcal{S}_b}}(|v|)$ are the destinations of the outgoing transitions of $\mathcal{R}_b$ and $\overline{\mathcal{S}_b}$ by construction. Indeed, it must exist an outgoing transition of the intersection automaton that corresponds to the synchronous execution of the outgoing transitions of $\mathcal{R}_b$ and $\overline{\mathcal{S}_b}$. Thus, the run $\rho_{\mathcal{I}}$ is a finite accepting run for the intersection automaton and $v \notin \mathcal{L}^{i*}(\mathcal{I})$ that contradicts the hypothesis. □

Lemma 5.3.2 (Finite external intersection language)**.** *The intersection automaton $\mathcal{I} = \mathcal{R}_b \cap \overline{\mathcal{S}_b}$ between the replacement $\mathcal{R}_b$ and the sub-property $\overline{\mathcal{S}_b}$ recognizes the finite external language $\mathcal{L}^{e*}(\mathcal{I}) = (\mathcal{L}^{e*}(\mathcal{R}_b) \cup \mathcal{L}_p^{e*}(\mathcal{R}_b)) \cap \mathcal{L}^{e*}(\overline{\mathcal{S}_b})$, i.e., $v \in \mathcal{L}^{e*}(\mathcal{I}) \Leftrightarrow v \in ((\mathcal{L}^{e*}(\mathcal{R}_b) \cup \mathcal{L}_p^{e*}(\mathcal{R}_b)) \cap \mathcal{L}^{e*}(\overline{\mathcal{S}_b}))$.*

Lemma 5.3.3 (Infinite internal intersection language)**.** *The intersection automaton $\mathcal{I} = \mathcal{R}_b \cap \overline{\mathcal{S}_b}$ between the replacement $\mathcal{R}_b$ and the sub-property $\overline{\mathcal{S}_b}$ recognizes the infinite internal language $\mathcal{L}^{i\omega}(\mathcal{I}) = (\mathcal{L}^{i\omega}(\mathcal{R}_b) \cup \mathcal{L}_p^{i\omega}(\mathcal{R}_b)) \cap \mathcal{L}^{i\omega}(\overline{\mathcal{S}_b})$, i.e., $v^\omega \in \mathcal{L}^{i\omega}(\mathcal{I}) \Leftrightarrow v \in ((\mathcal{L}^{i\omega}(\mathcal{R}_b) \cup \mathcal{L}_p^{i\omega}(\mathcal{R}_b)) \cap \mathcal{L}^{i\omega}(\overline{\mathcal{S}_b}))$.*

Lemma 5.3.4 (Infinite external intersection language)**.** *The intersection automaton $\mathcal{I} = \mathcal{R}_b \cap \overline{\mathcal{S}_b}$ between the replacement $\mathcal{R}_b$ and the sub-property $\overline{\mathcal{S}_b}$ recognizes the infinite external language $\mathcal{L}^{e\omega}(\mathcal{I}) = (\mathcal{L}^{e\omega}(\mathcal{R}_b) \cup \mathcal{L}_p^{e\omega}(\mathcal{R}_b)) \cap \mathcal{L}^{e\omega}(\overline{\mathcal{S}_b})$, i.e., $v^\omega \in \mathcal{L}^{e\omega}(\mathcal{I}) \Leftrightarrow v \in ((\mathcal{L}^{e\omega}(\mathcal{R}_b) \cup \mathcal{L}_p^{e\omega}(\mathcal{R}_b)) \cap \mathcal{L}^{e\omega}(\overline{\mathcal{S}_b}))$.*

Proof. The proofs of Lemmas 5.3.2, 5.3.3 and 5.3.4 can be easily derived from the proof of Lemma 5.3.1. □

The *under approximation* automaton is used by an emptiness checking procedure to verify whether the claim *is not* satisfied, i.e., it encodes the behaviors that violate the property of interest. The automaton is computed exploiting the information of the lower reachability relation $\aleph_c$ and in the Π function.

Definition 5.3.3 (Under approximation automaton)**.** *Given the sub-property $\overline{\mathcal{S}_b} = \langle \overline{\mathcal{P}_b}, \Delta^{in\overline{\mathcal{S}_b}}, \Delta^{out\overline{\mathcal{S}_b}}, \Pi, \aleph, \aleph_c \rangle$ associated to the box b, the replacement $\mathcal{R}_b = \langle \mathcal{M}_b, \Delta^{inR_b},$*

$\Delta^{outR_b}\rangle$, *two additional automata states* g *and* r*, the under approximation automaton* $\mathcal{E}_c$ *is the automaton obtained from* $\mathcal{I}_c$ *as follows:*

- $\Sigma_{\mathcal{E}_c} = \Sigma_{\mathcal{I}_c}$*;*
- $Q_{\mathcal{E}_c} = Q_{\mathcal{I}_c} \cup \{g, r\}$*;*
- $\Delta_{\mathcal{E}_c} = \Delta_{\mathcal{I}_c} \cup \Delta^{in}_{\mathcal{E}_c} \cup \Delta^{out}_{\mathcal{E}_c} \cup \Delta^{\aleph_c}_{\mathcal{E}_c} \cup \Delta^{stut}_{\mathcal{E}_c}$*, where*
  - $\Delta^{stut}_{\mathcal{E}_c} = \{(r, stut, r)\}$*;*
  - $\Delta^{in}_{\mathcal{E}_c} = \{(g, a, s') \mid (s, a, s') \in \Delta^{inI_b}$ *and* $\Pi((s, a, s')) = G\}$*;*
  - $\Delta^{out}_{\mathcal{E}_c} = \{(s, a, r) \mid (s, a, s') \in \Delta^{outI_b}$ *and* $\Pi((s, a, s')) = R\}$*;*
  - $\Delta^{\aleph_c}_{\mathcal{E}_c} = \{(\langle q, p, x\rangle, \epsilon, \langle q', p', y\rangle) \mid ((q, a, q''), (q''', b, q')) \in \aleph_c\}$*. Moreover, the values* x *and* y *associated with* $\delta_o = (q, a, q'')$*, and* $\delta_i = (q''', b, q')$ *must satisfy the following conditions:*
    - *if* $\Gamma^{\aleph_c}_{\mathcal{M}}(\delta_o, \delta_i) = T$ *and* $\Gamma^{\aleph_c}_{\overline{\mathcal{A}_\phi}}(\delta_o, \delta_i) = T$ *then* $y = 2$*;*
    - *else if* $x = 1$ *and* $\Gamma^{\aleph_c}_{\overline{\mathcal{A}_\phi}}(\delta_o, \delta_i) = T$ *or* $p' \in F_{\overline{\mathcal{P}}_b}$ *then* $y = 2$*;*
    - *else if* $x = 0$ *and* $\Gamma^{\aleph_c}_{\mathcal{M}}(\delta_o, \delta_i) = T$ *and* $p' \in F_{\overline{\mathcal{P}}_b}$ *then* $y = 2$*;*
    - *else if* $x = 0$ *and* $\Gamma^{\aleph_c}_{\mathcal{M}}(\delta_o, \delta_i) = T$ *or* $q' \in F_{\mathcal{R}_b}$ *then* $y = 1$*;*
    - *else if* $x = 2$ *then* $y = 0$*;*
    - *else* $y = x$*.*
- $Q^0_{\mathcal{E}_c} = Q^0_{\mathcal{I}_c} \cup \{g\}$*;*
- $F_{\mathcal{E}_c} = F_{\mathcal{I}_c} \cup \{r\}$*.*

The completion of the extended intersection automaton contains all the behaviors of the intersection automaton that violate the claim ϕ plus additional transitions which specify how these behaviors are related to each others. The state g is used as a placeholder to represent the initial states of the system and the transitions in $\Delta^{in}_{\mathcal{E}_c}$ specify how the states of the intersection are reachable from the initial states. Similarly, the state r and the transition in $\Delta^{stut}_{\mathcal{E}_c}$ are used as placeholders for a suffix of a run that does not involve the replacement of boxes and violates the claim. The transitions in $\Delta^{out}_{\mathcal{E}_c}$ specify how it is possible to reach this violating run from the intersection between the sub-property and the refinement. Finally, the transitions in $\Delta^{\aleph_c}_{\mathcal{E}_c}$ specify how the violating behaviors of the intersection automaton between the replacement and the sub-property (which are portions of the intersection automaton between the refinement and the property) influence each other. Note that, as done in the computation of the classical intersection automaton, it is necessary to compute the value of y in the intersection state $\langle q', p', y\rangle$. The value of y depends on the presence of accepting states of the replacement and the sub-property over the runs made by purely regular states that connect the outgoing to the incoming transitions of the replacement, i.e., on the functions $\Gamma^{\aleph_c}_{\mathcal{M}}$, $\Gamma^{\aleph_c}_{\overline{\mathcal{A}_\phi}}$. More precisely, y is identified as follows: whenever there exists both an accepting state of the model and of the claim in the original intersection automaton in a run made by purely regular states that connects the outgoing transition δ_o and the incoming transition δ_i, the value of y is 2 to force the presence of an accepting state in the run. Similarly, if the value of x is equal to 1 and there exists a run in the intersection

automaton that connects δ_o to δ_i which traverses an accepting state of $\overline{\mathcal{P}_b}$, the value of y is set to 2 to force the presence of an accepting state in the run. Finally, $y = 2$ also if the value of x is equal to 0 there exists a run in the intersection automaton that connects δ_o to δ_i which traverses an accepting state of the model $\mathcal{M}$ and the destination state is accepting for the sub-property. Otherwise, if the value of x is equal to 0 and there exists a run in the intersection automaton that connects δ_o to δ_i which traverses an accepting state of the model $\mathcal{M}$ or the destination state is an accepting state of the replacement $\mathcal{R}_b$ then $y = 1$. If $x = 2$, then $y = 0$. In the other cases $y = x$.

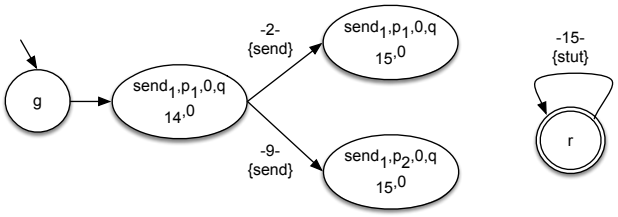


Figure 5.8: *The under approximation of the intersection described in Figure 5.7.*

For example, the under approximation of the intersection automaton described in Figure 5.7 is presented in Figure 5.8. Note that the reachability relation does not cause the injection of any transition in the intersection automaton. Furthermore, the accepting state marked with r is not reachable.

The *over* approximation of the intersection automaton is similar to the under approximation, but the upper reachability relation $\aleph$ and the incoming and outgoing transitions also marked with a Y symbol through the function Π are considered. This because the over approximation is the automaton to be used by the emptiness checking procedure to verify whether the claim *is possibly* satisfied.

Definition 5.3.4 (Over approximation automaton)**.** *Given the sub-property* $\overline{\mathcal{S}_b} = \langle \overline{\mathcal{P}_b}, \Delta^{in\overline{\mathcal{S}_b}}, \Delta^{out\overline{\mathcal{S}_b}}, \Pi, \aleph, \aleph_c \rangle$ *associated with the box* b*, and the replacement* $\mathcal{R} = \langle \mathcal{M}_b, \Delta^{inR_b}, \Delta^{outR_b} \rangle$*, the extended intersection automaton* $\mathcal{E}$ *is the automaton obtained from* $\mathcal{I}$ *such as:*

- $\Sigma_{\mathcal{E}} = \Sigma_{\mathcal{I}}$*;*
- $Q_{\mathcal{E}} = Q_{\mathcal{I}} \cup \{g, r, y_i, y_a\}$*;*
- $\Delta_{\mathcal{E}} = \Delta_{\mathcal{I}} \cup \Delta^{in}_{\mathcal{E}} \cup \Delta^{out}_{\mathcal{E}} \cup \Delta^{\aleph_c}_{\mathcal{E}} \cup \Delta^{stut}_{\mathcal{E}}$*, where*
  - $\Delta^{stut}_{\mathcal{E}} = \{(r, stut, r), (y_a, stut, y_a)\}$*;*
  - $\Delta^{in}_{\mathcal{E}} = \Delta^{in}_{\mathcal{E}_c} \cup \{(y_i, a, s') \mid (s, a, s') \in \Delta^{inI_b}$ *and* $\Pi(s, a, s') = Y\}$*;*
  - $\Delta^{out}_{\mathcal{E}} = \Delta^{out}_{\mathcal{E}_c} \cup \{(s, a, y_a) \mid (s, a, s') \in \Delta^{outI_b}$ *and* $\Pi(s, a, s') = Y\}$*;*
  - *and* $\Delta^{\aleph}$ *is defined as* $\Delta^{\aleph_c}$ *with the exception that the upper reachability relation* $\aleph$ *and the functions* $\Gamma^{\aleph}_{\mathcal{M}}$*,* $\Gamma^{\aleph}_{\overline{\mathcal{A}_\phi}}$ *are considered;*
- $Q^0_{\mathcal{E}} = Q^0_{\mathcal{I}} \cup \{g, y_i\}$*;*
- $F_{\mathcal{E}} = F_{\mathcal{I}} \cup \{r, y_a\}$*;*

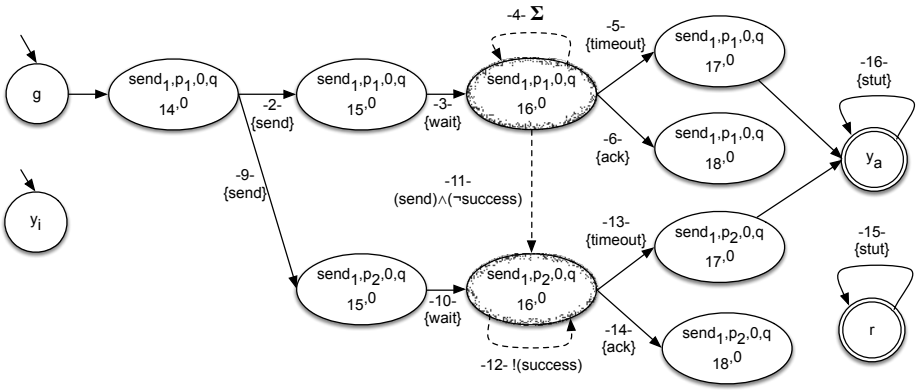


Figure 5.9: *The over approximation of the intersection automaton described in Figure 5.7.*

Since the extended intersection automaton contains all the behaviors of the intersection automaton that violate and *possibly* violate the claim ϕ, it includes all the runs that connect an incoming transition marked as G or Y with an outgoing marked as R or Y. These runs may include transitions of the intersection automaton (which can be also generated by other boxes of the model), or transitions of the reachability graph, which are used to abstract runs of the intersection between $\mathcal{M}$ and $\overline{\mathcal{A}_\phi}$ made by only purely regular and mixed states.

A portion of the over approximation of the intersection automaton described in Figure 5.7 is presented in Figure 5.9. Note that the state y_a which describes the presence of a suffix of a possibly violating is reachable from the intersection automaton.

5.3.2 The model checking procedure

The replacement checking procedure, given the constraint $\mathcal{C}$ and a replacement $\mathcal{R}_b$, checks whether $\mathcal{R}_b$ satisfies, possibly satisfies or does not satisfy $\mathcal{C}$. The replacement $\mathcal{R}_b$ *does not satisfy* the constraint $\mathcal{C}$ if and only if the under approximation of the intersection automaton obtained considering the sub-property $\overline{\mathcal{S}_b}$ associated to the box b and the replacement $\mathcal{R}_b$ is not empty. The replacement $\mathcal{R}_b$ *possibly satisfies* the constraint $\mathcal{C}$ if and only if the over approximation of the intersection automaton is not empty or $\mathcal{U}(b) = T$, otherwise, the constraint is satisfied. Formally,

Definition 5.3.5 (Replacement checking)**.** *Given the constraint $\mathcal{C} = \langle \zeta, \mathcal{U} \rangle$, the sub-property $\overline{\mathcal{S}_b} = \langle \overline{\mathcal{P}_b}, \Delta^{in\overline{\mathcal{S}_b}}, \Delta^{out\overline{\mathcal{S}_b}}, \Pi, \aleph, \aleph_c, \Gamma^{\kappa}_{\mathcal{M}}, \Gamma^{\kappa}_{\overline{\Phi}} \rangle$, such that $\overline{\mathcal{S}_b} \in \zeta$ associated to the box b, and the replacement $\mathcal{R}_b = \langle \mathcal{M}_b, \Delta^{inR_b}, \Delta^{outR}_s \rangle$,*

1. $\|\mathcal{R}^{\mathcal{C}}_b\| = F \Leftrightarrow \mathcal{L}(\mathcal{E}_c) \neq \emptyset$;
2. $\|\mathcal{R}^{\mathcal{C}}_b\| = T \Leftrightarrow \mathcal{L}(\mathcal{E}) = \emptyset \wedge \mathcal{U}(b) = F$;
3. $\|\mathcal{R}^{\mathcal{C}}_b\| = \perp \Leftrightarrow \|\mathcal{R}^{\mathcal{C}}\| \neq F \wedge \|\mathcal{R}^{\mathcal{C}}\| \neq T$.

where $\mathcal{E}_c$ and $\mathcal{E}$ are the automata obtained as specified in Definitions 5.3.3 and 5.3.4.

The idea behind the model checking procedure proposed in this section is to reduce the model checking problem to a cycle detection problem. A similar idea has been

used, for example, in the model checking of Hierarchical Kripke structures [8]. To demonstrate the correctness of our definition, we prove that checking a replacement $\mathcal{R}_b$ versus its constraint $\mathcal{C}$ corresponds to checking the refined automaton $\mathcal{N}$ against the property ϕ.

Theorem 5.3.1 (Replacement checking correctness)**.** *Given a model $\mathcal{M}$, a property ϕ, a replacement $\mathcal{R}_b$ for a box b and the constraint $\mathcal{C}$ obtained as previously described:*

1. $\|\mathcal{R}_b^{\mathcal{C}}\| = F \Leftrightarrow \|\mathcal{N}^{\overline{\mathcal{A}_\phi}}\| = F$;
2. $\|\mathcal{R}_b^{\mathcal{C}}\| = T \Leftrightarrow \|\mathcal{N}^{\overline{\mathcal{A}_\phi}}\| = T$;
3. $\|\mathcal{R}_b^{\mathcal{C}}\| = \bot \Leftrightarrow \|\mathcal{N}^{\overline{\mathcal{A}_\phi}}\| = \bot$.

Proof. Let us starts by proving condition 1.

($\Rightarrow$) If $\|\mathcal{R}_b^{\mathcal{C}}\| = F$ by Definition 5.3.5 condition 1 it must exist a word v accepted by the automaton $\mathcal{E}_c$. Let us consider the run $\rho^\omega_{\mathcal{E}_c}$ associated with v. We want to generate a run $\rho^\omega_{\mathcal{I}}$ in the intersection automaton $\mathcal{I}$ between the model $\mathcal{N}$ and the claim $\overline{\mathcal{A}_\phi}$ which corresponds to $\rho^\omega_{\mathcal{E}_c}$. Let us consider the initial state $\rho^\omega_{\mathcal{E}_c}(0)$ of the run $\rho^\omega_{\mathcal{E}_c}$. Two cases are possible: *a*) $\rho^\omega_{\mathcal{E}_c}(0)$ is obtained by combining an initial state p of the automaton $\overline{\mathcal{P}_b}$ of the sub-property $\overline{\mathcal{S}_b}$ and an initial state q of the replacement $\mathcal{R}_b$. Note that the initial state p of the sub-property was obtained by combining an initial state p' of the property $\overline{\mathcal{A}_\phi}$ with a state q' of $\mathcal{M}$ which must be both initials. Furthemore, q' must be a box from construction (see Definitions 5.1.1 and 5.2.3). Since the refinement $\mathcal{N}$ contains all the states of the replacement $\mathcal{R}_b$ and an initial state of $\mathcal{R}_b$ is also initial for $\mathcal{N}$, it is possible to associate to $\rho^\omega_{\mathcal{I}}(0)$ the state $\langle q', p, 0\rangle$ of $\mathcal{I}$. *b*) $\rho^\omega_{\mathcal{E}_c}(0)$ corresponds to the state g. Consider a transition $\delta \in \Delta^{in}_{\mathcal{E}_c}$ that starts from the g state. The transition δ is obtained by combining a transition $\delta^{inR_b} \in \Delta^{inR_b}$ with a transition $\delta^{in\overline{\mathcal{S}_b}} \in \Delta^{in\overline{\mathcal{S}_b}}$, which is in turn obtained by combining a transition $\delta_{\mathcal{M}} \in \Delta_{\mathcal{M}}$ and a transition $\delta_{\overline{\mathcal{A}_\phi}} \in \Delta_{\overline{\mathcal{A}_\phi}}$. Let us consider the source states $q_{\mathcal{M}}$ and $p_{\mathcal{M}}$ of the transitions $\delta_{\mathcal{M}}$ and $\delta_{\overline{\mathcal{A}_\phi}}$ it is possible to replicate the run that reaches these states in the intersection automaton $\mathcal{I}$ since by definition plugging a replacement (Definition 4.2.8) does not modify behaviors in which only regular states are involved. Furthermore, the transition obtained from δ^{inR_b} and $\delta^{in\overline{\mathcal{S}_b}}$ can be associated with the transition of the intersection automaton obtained combining δ^{inR_b} and $\delta_{\overline{\mathcal{A}_\phi}}$. Let us now consider the other states of the run. Each state of the under approximation automaton can be rewritten as $\langle q_{\mathcal{R}}, \langle b, p, x\rangle, y\rangle$ since it is obtained by combining a state of the sub-property, which has the form $\langle b, p, x\rangle$, with a state $q_{\mathcal{R}}$ of the replacement. Each of these states can be associated with the state $\langle q_{\mathcal{R}}, p, y\rangle$ of the intersection between the replacement and the sub-property. Similarly, each transition $\delta_{\mathcal{I}} \in \Delta_{\mathcal{I}}$ of the intersection between the replacement and the sub-property is obtained by firing a transition of the replacement and an internal transition of the sub-property which corresponds to a transition of the original property, i.e., in $\Delta_{\overline{\mathcal{A}_\phi}}$, and a transition of the model, i.e., in $\Delta_{\mathcal{M}}$. Thus, the same transition can be identified in the intersection automaton obtained considering $\mathcal{N}$ and $\overline{\mathcal{A}_\phi}$. Let us finally consider a transition $(\rho_{\mathcal{E}_c}(i), a, \rho_{\mathcal{E}_c}(i+1)) \in (\Delta^{\aleph_c}_{\mathcal{E}_c} \cup \Delta^{stut}_{\mathcal{E}_c})$ from construction (see Definitions 5.2.5) it must exists a sequence of transitions in the automaton $\mathcal{I}$ obtained from $\mathcal{N}$ and $\overline{\Phi}$ that connect only purely regular states and reach an accepting state that can be entered infinitely often and corresponds to this transition.

($\Leftarrow$) The proof is by contradiction. Let us assume that $\|\mathcal{R}_b^{\mathcal{C}}\| \neq F$ and $\|\mathcal{N}^{\overline{\mathcal{A}_\phi}}\| = F$. Since $\|\mathcal{N}^{\overline{\mathcal{A}_\phi}}\| = F$, it must exist a word v accepted by the automaton $\mathcal{I}$ obtained from $\mathcal{N}$ and $\overline{\mathcal{A}_\phi}$. Since this run must be accepting, it must involve only purely regular states of $\mathcal{I}$. However, since v was not accepted by the intersection obtained from $\mathcal{M}$ and $\overline{\mathcal{A}_\phi}$, some of these states must obviously be obtained by combining states of $\mathcal{R}_b$ and of $\overline{\mathcal{A}_\phi}$. This implies the presence of an accepting run in the automaton $\mathcal{E}_c$, which may connect the state "g" with the state "r" or another accepting state of $\mathcal{E}_c$ that can be entered infinitely often. Thus, $\|\mathcal{R}_b^{\mathcal{C}}\| \neq F$ is contradicted.

Let us now consider the condition 2 of Theorem 5.3.1. The proof corresponds to the one proposed for 1, but, in this case, the mixed states of the intersection automaton and the upper reachability relation $\aleph$ are considered. Furthermore, if the function $\mathcal{U}(b) = T$, the sub-property is possible satisfied. Indeed, in this case, there exists a possibly accepting run in the intersection between the model $\mathcal{M}$ and the property $\overline{\mathcal{A}_\phi}$ that does not depend on the refinement of b. Thus, ϕ is possibly satisfied since the same run will be present in the intersection between $\mathcal{N}$ and the property $\overline{\mathcal{A}_\phi}$.

The proof of condition 3 of Theorem 5.3.1 follows from the proofs of conditions 1 and 2. □

Lemma 5.3.5 (Checking a replacement complexity)**.** *The complexity of the model checking procedure is proportional to the size of the automata $\mathcal{E}$ and $\mathcal{E}_c$, which in the worst case is* $\mathcal{O}(|Q_{\mathcal{R}_b}|\cdot|Q_{\overline{\mathcal{P}_b}}|+|\Delta_{\mathcal{R}_b}|\cdot|\Delta_{\overline{\mathcal{P}_b}}|+|\Delta^{inR_b}|\cdot|\Delta^{in\overline{\mathcal{S}_b}}|+|\Delta^{outR_b}|\cdot|\Delta^{out\overline{\mathcal{S}_b}}|+(|\Delta^{out\overline{\mathcal{S}_b}}|\cdot|\Delta^{in\overline{\mathcal{S}_b}}|)\cdot(|\Delta^{outR_b}|\cdot|\Delta^{inR_b}|))$

Proof. The size of the automata described in Lemma 5.3.5 is justified by the following statements. The size of the automaton obtained by considering the automaton $\mathcal{M}_b$ associated with the replacement $\mathcal{R}_b$ of the box b and the automaton $\overline{\mathcal{P}_b}$ associated with the sub-property $\overline{\mathcal{S}_b}$ contains in the worst case $|Q_{\mathcal{R}_b}| \cdot |Q_{\overline{\mathcal{P}_b}}|$ states and $|\Delta_{\mathcal{R}_b}| \cdot |\Delta_{\overline{\mathcal{P}_b}}|$ transitions. This automaton can be reached through a set of transitions which are obtained by the synchronous execution of an incoming transition of the replacement and the sub-property, leading in the worst case to $|\Delta^{inR_b}| \cdot |\Delta^{in\overline{\mathcal{S}_b}}|$ transitions. Similarly, the automaton can be left through a set of transitions obtained by the synchronous execution of an outgoing transition of the replacement and of the sub-property generating in the worst case $|\Delta^{outR_b}| \cdot |\Delta^{out\overline{\mathcal{S}_b}}|$ transitions. Finally, each pair outgoing/incoming transition contained in the reachability relation of the sub-property can be synchronized with every pair outgoing/incoming transition of the replacement, leading to $(|\Delta^{out\overline{\mathcal{S}_b}}| \cdot |\Delta^{in\overline{\mathcal{S}_b}}|) \cdot (|\Delta^{outR_b}| \cdot |\Delta^{inR_b}|)$ transitions. □

CHAPTER 6

Automated Tool Support

"I have been impressed with the urgency of doing. Knowing is not enough; we must apply. Being willing is not enough; we must do."

Leonardo da Vinci, 1452-1519

This section presents `CHIA` (CHecker for Incompete Automata) a prototype tool[1] realized as a `Java 7` stand-alone application. The tool has been developed as a proof of concepts and does not aim to compete with state of the art model checking tools. It provides a command-line shell which allows the developer to *a*) load the models, the properties, the constraints and the replacements of interest; *b*) check the incomplete models against the corresponding properties; *c*) in the cases in which the property is possibly satisfied, compute a constraint for the unspecified parts; *d*) check the replacement against the corresponding constraints. The tool is developed as a `Maven` multi-module project. It is composed by different modules which encapsulate different parts of the `CHIA` logic. Section 6.1 presents an overview of the `CHIA` framework. Section 6.2 illustrates the `CHIAAutomata` module which contains the classes used to explicitly model the state space of the automata. Section 6.3 describes the model checker that bears the verification of the model expressed against the properties of interest. Section 6.4 presents the `CHIAConstraint` module. This module includes the classes which describe a constraint and the corresponding sub-properties. Section 6.5 specifies how the constraint is computed, while Section 6.6 contains the module which allows to check a replacement against a previously generated constraint.

[1] The tool is available at `http://home.deib.polimi.it/menghi/Tools/IncModChk.html`.

6.1 Overall framework

The `CHIAFramework` is the main module of `CHIA`. It contains the entry point for the use of the framework, i.e., the method to run the command line shell. The command line shell is implemented using the `jline` library [1], which allows to create interactive command-line user interfaces.

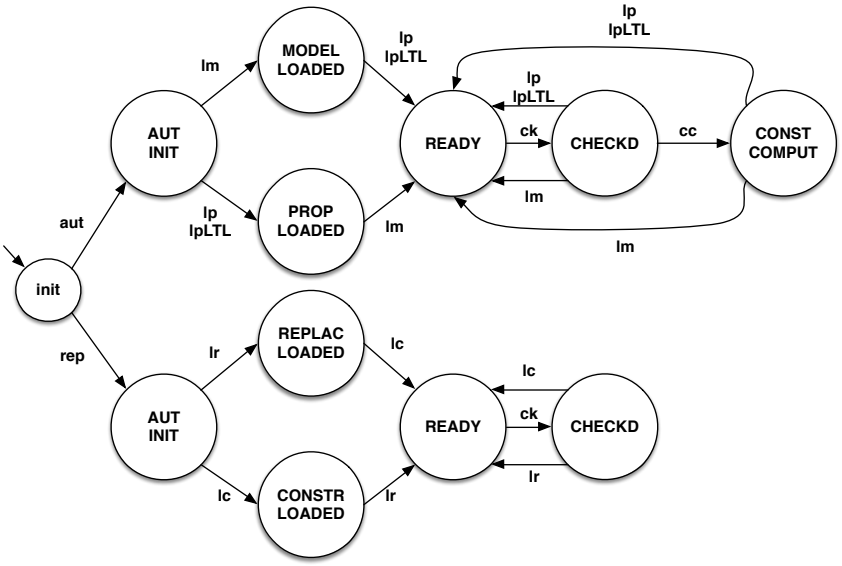


Figure 6.1: *The state machine describing the behavior of the `CHIA` framework.*

The state of `CHIA` changes in response to the user requests as specified in Figure 6.1. The user can first select the modality of interest, i.e., if he/she wants to use the automata checker (`aut`) or the replacement checker (`rep`). In the automata mode the user is able to load the property from an automaton saved in an appropriate file (`lp`) or generating the automaton from an LTL formula (`lpLTL`). Similarly, the model of the system is loaded from an appropriate file, that contains the corresponding automaton, through the command `lm`. After both the model and the property have been loaded, the developer may verify (`ck`) if the system possesses the properties of interest. If the property is possibly satisfied the `cc` command allows the computation of the corresponding constraint. Whenever the replacement checking mode is activated (`rep`), the developer can load the replacement (`lr`) and the corresponding constraint (`lc`). The `ck` command verifies whether the refinement of the automaton possesses the properties of interest as specified in Section 5.3.

The classes that support the state machine described in Figure 6.1 are contained in different `Maven` modules.

6.2 Automata module

The `CHIAAutomata` module contains the classes which are used to manage BAs and IBAs, i.e., the `BA` and `IBA` class. These two classes describe BAs and IBAs using an explicit representation of the state space, which is build upon the `JGraphT` library [94].

The class diagram of the `CHIAAutomata` module is represented in Figure 6.2 and is hereafter discussed:

- `State`. It is used to represent a state of an automaton. Each state has two *final* attributes, the `id` and the `name` which are returned through the corresponding methods `int getId()` and `String getName()`.

- `StateFactory`. It is used for creating the states of the automaton. It assign an auto-generated id to the state, if it is not explicitly defined by the developer. It implements the `VertexFactory` interface of `JGraphT`.

- `Transition`. It is used to represent a transition of an automaton (BA or IBA). The transition is identified by an `id` and it is labeled with a `Set` of propositions, which are the propositions to be true for the transition to be fired. These propositions are represented using the `IGraphProposition` interface of the `LTL2BA4J` [3] tool. This choice allows the easy integration of the `LTL2BA4J` library, which supports the conversion of LTL formulae into the corresponding BAs. Note that this representation also allows a proposition to be negated, i.e., to represent a condition that labels a transition of the property. The set of propositions (negated or not) added to the transitions are considered as being connected by a logical `AND`. The `Transition` class which represents the transitions of the automata extends the `DefaultEdge` class of `JGraphT`. The id and the propositions are returned through the methods `int getId()` and `Set<IGraphProposition> getPropositions()`, respectively.

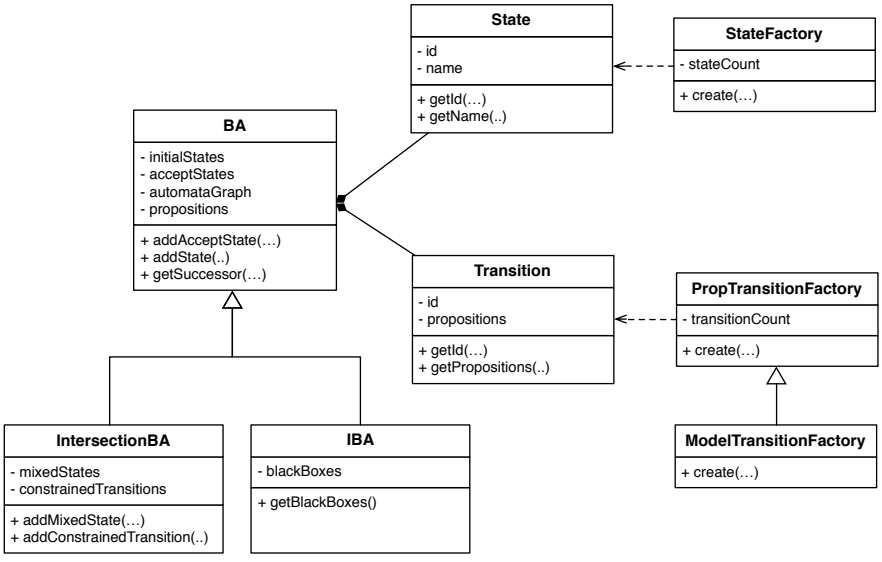


Figure 6.2: *The class diagram of the `CHIAAutomata` module.*

- `ModelTransitionFactory`. It is used to create the transitions of the model. When the factory creates a transition of the model it checks that no negated propositions are used to decorate the transition, i.e., the propositions specify statements that are true when the transition is performed.

- `PropTransitionFactory`. It creates the transitions of the properties. In the claim case, also negated propositions are allowed since the transition encodes a condition (propositional logic formula) that must be satisfied for the transition to be fired.
- `BA`. It contains the attributes and methods that describe a Büchi Automaton. The `automataGraph` attribute of the `BA` class represents the graph upon which the automaton is built. The graph is modeled using the `DirectedPseudograph` class of the `JGraphT` library. The `DirectedPseudograph` class allows the creation of a non-simple directed graph in which both *loops* and *multiple edges* are permitted. The `BA` attributes include the set of initial states, the set of accepting states, and the alphabet of the Büchi Automaton. The constructor of the `BA` requires to specify the transition factory to be used. Depending on whether the `BA` is used to represent the model or the claim to be checked a different factory can be specified. The `BA` class provides methods to access the states of the BA, such as the one that allow to get the successors of a state (`getSuccessors()`) or the initial states (`getInitialStates()`) of the BA.
- `IBA`. It models an Incomplete Büchi Automaton. The `IBA` class extends the `BA` class. The `blackBoxes` attribute is used to store the set of states of the IBA which are *black box*. The black box states are returned by the method `getBlackBoxes()`.
- `IntersectionBA`. It contains the automaton obtained from the intersection between a BA and an IBA. The `IntersectionBA` class extends `BA`. The `mixedStates` attribute is used to store the set of mixed states. The mixed states are obtained by considering a black box state of the model and a state of the property as specified in Section 5.1.1. The `constrainedTransition` attribute is used to store the transitions generated by performing a transition of the property inside a box of the model as specified in Section 5.1.1.

6.2.1 Automata input/output module

The `CHIAAutomataIO` module provides the classes to load and save BAs and IBAs from and to an appropriate `XML` file. The module uses the classes available in the `javax.xml.parsers` package of `Java` instead of using existing libraries, such as `JAXB` [2] or the `GraphMLExporter` class of `JGraphT`. This choice has been performed to provide a higher flexibility and customization of the I/O Files. A portion of the class diagram corresponding to the `CHIAAutomataIO` module, which refers to the `BA` class, is presented in Figure 6.3. The same classes are also present for the `IBA` class with a different implementation.

The classes presented in Figure 6.3 are designed to load and save a BA from the corresponding XML file:

- `PropReader`. It is used to load a `BA` from the corresponding `XML` representation. It is based on the `ElementToBATransformer` class which transforms an `XML` element into the corresponding BA.
- `ElementToBATransformer`. It is used to transform an `XML` element which represents a BA into the corresponding object.

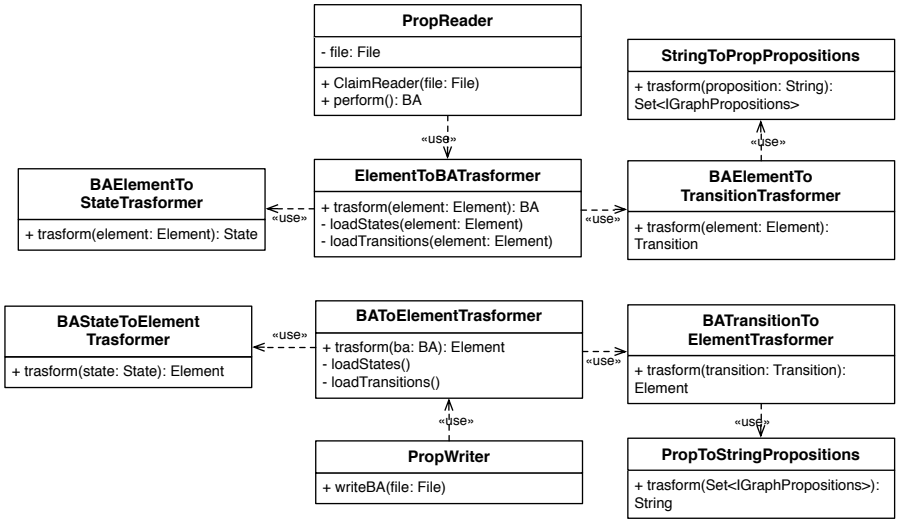


Figure 6.3: *A portion of the class diagram of the* `CHIAAutomataIO` *module.*

- `BAElementToStateTransformer`. It converts an `XML` element which represents a BA state into the corresponding `State` object.
- `BAElementToTransitionTransformer`. It is used to transform an `XML` element which represents a BA transition into the corresponding `Transition` object.
- `StringToPropPropositions`. It converts a `String` which specifies the propositions that decorate the transition into the corresponding `Set` of `IGraphProposition`.
- `PropWriter`. It is used to write a `BA` to the corresponding `XML` file. It uses the `BAToElementTrasformer` which transforms a BA object into the corresponding `XML` element.
- `BAToElementTrasformer`. It is used to transform a `BA` into the corresponding `XML` element.
- `BAStateToElementTrasformer`. It converts a `State` object into the corresponding `XML` representation.
- `BATransitionToElementTrasformer`. It is used to transform a `Transition` object into the corresponding `XML` representation.
- `PropToStringPropositions`. It converts a set of propositions into the corresponding `String` representation.

The structure of the XML files that contain the Büchi and the Incomplete Büchi automata is specified in the `BA.xsd` and the `IBA.xsd` files.

Listing 6.1 contains the XML file of the model presented in Figure 5.1a. The `XML` file is composed by three different parts delimited by the `XML` tags `<propositions>`, `<states>` and `<transitions>` which contain the propositions, the states and the transitions of the IBA, respectively.

The name of each proposition of the automaton is specified into the `name` attribute of the `XML proposition` element. The proposition must be specified using lower case identifiers.

Each state of the Büchi automaton is represented in a `state` XML element. The `id` and the `name` attributes contain the id and the name of the state, respectively. If the state is initial, accepting or is a black box the corresponding attributes are set to the `true` value.

Each transition of the automaton is represented through an XML `transition` element. The `id`, the `source` and the `destination` attributes contain the id of the transition and the ids of the source and the destination states. The `propositions` attribute contains a `String` which represent a conjunction of propositions which are true when the transition is performed.

```
<iba>
  <propositions>
    <proposition name="ok"/>
    <proposition name="abort"/>
    <proposition name="fail"/>
    <proposition name="start"/>
    <proposition name="success"/>
  </propositions>
  <states>
    <state id="1" name="q1" initial="true"/>
    <state id="2" name="send1" blackbox="true"/>
    <state id="3" name="send2" blackbox="true"/>
    <state id="4" name="q2" accepting="true"/>
    <state id="5" name="q3" accepting="true"/>
  </states>
  <transitions>
    <transition id="1" source="1" destination="2" propositions="start"/>
    <transition id="2" source="2" destination="3" propositions="fail"/>
    <transition id="3" source="2" destination="5" propositions="ok"/>
    <transition id="4" source="3" destination="5" propositions="ok"/>
    <transition id="5" source="3" destination="4" propositions="fail"/>
    <transition id="6" source="4" destination="4" propositions="abort"/>
    <transition id="7" source="5" destination="5" propositions="success"/>
  </transitions>
</iba>
```

Listing 6.1: *The XML file corresponding to the model presented in Figure 5.1a*

The XML file used to load the claim, i.e., a BA, has the same structure of the one presented in Listing 6.1 with the following exceptions: *a*) a state cannot be a black box; *b*) the `propositions` attribute of the transitions can also contain negated propositions.

6.3 Model checker

The `CHIAChecker` module contains the classes to check whether a model described through an `IBA` *satisfies*, *does not satisfy* or *possibly satisfies* a property of interest. The class diagram corresponding to the `CHIAChecker` module is presented in Figure 6.4.

- `SatisfactionValue`. It is an enumeration that contains the possible results of the model model checking activity: `SATISFIED`, `NOTSATISFIED` or `POSSIBLYSATISFIED`.
- `Checker`. It contains the entry point used to run the model checking tool. It requires as an input a `BA` and an `IBA`. The `perform` method returns

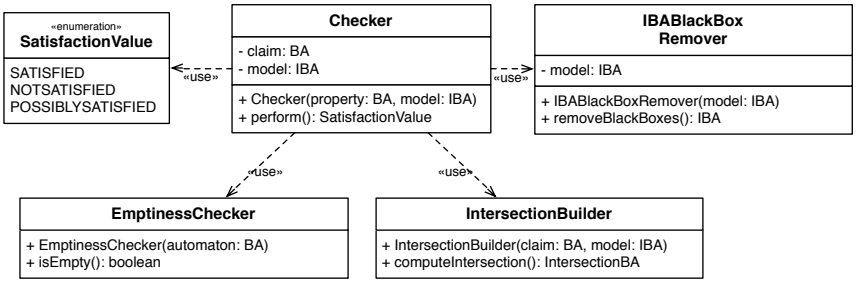


Figure 6.4: *The class diagram of the* `CHIAChecker` *module.*

one of the satisfaction values depending on the checking result. The checker uses the `IBABlackBoxRemover`, the `IntersectionBuilder` and the `EmptinessChecker` classes.

- `IBABlackBoxRemover`. It is used to remove the black box states and their incoming and outgoing transitions from the `IBA` used to represent the model. In particular, the method `removeBlackBoxes()` returns a copy of the `IBA` where the black box states are removed.

- `IntersectionBuilder`. It computes the intersection between a model (expressed as an `IBA`) and a property (expressed as a `BA`).

- `EmptinessChecker`. It checks the emptiness of an automaton. It implements the double DFS algorithm. The method `isEmpty()` returns `true` if the automaton is empty, i.e., if it does not exists an infinite run that contains an accepting state that can be accessed infinitely often, `false` otherwise.

6.4 Constraint module

The `CHIAConstraint` module contains the classes which are used to describe *constraints*, *subproperties* and *replacements*. The class diagram of the `CHIAConstraint` module is presented in Figure 6.5. The main components of the module are described in the following:

- `Component`. It is the abstract class which is used to describe components, i.e., replacements and sub-properties. Each component refers to a particular state which is represented by the final attribute `modelState`.

- `Replacement`. It is used to represent a replacement. It extends the `Component` class with the `IBA` which is used to refine the black box state and the set of its incoming and outgoing transitions which specifies how the replacement is connected to the original model. The incoming and outgoing transitions are modeled through the class `PluggingTransition`.

- `SubProperty`. It contains the description of the sub-property. The `Subproperty` class extends the `Component` class by specifying the `BA` which describes the claim the developer must consider in the refinement process. The set of the incoming and outgoing transitions associated with the

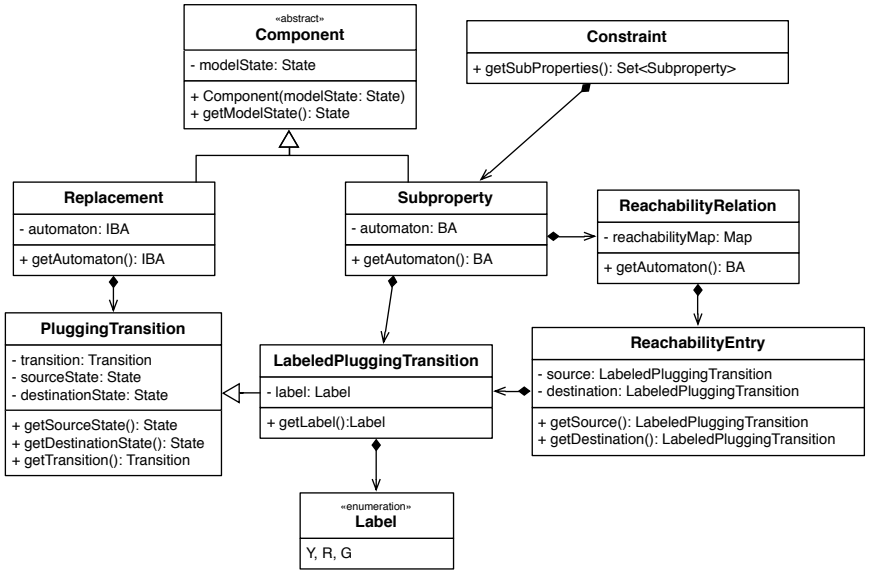


Figure 6.5: *The class diagram of the* `CHIAConstraint` *module.*

sub-property that specify how the sub-property is related with the original model. The incoming and outgoing transitions are specified using the class `LabeledPluggingTransition`. The reachability relation specifies the reachability between its incoming and outgoing transitions.

- `PluggingTransition`. It describes how the automaton that refers to the sub-property/replacement associated with a box is connected with the states of the original model. The `PluggingTransition` class contains the `source`, `destination` and `transition` attributes, i.e., the source and the destination state of the incoming/outgoing transition, and the transition itself. Depending on whether the `PluggingTransition` represents an incoming or an outgoing transition the source or the destination state correspond with a state of the model.

- `LabeledPluggingTransition`. It represents the incoming and outgoing transitions associated with a sub-property. As specified in Section 5.2.2 these transitions are also associated with a label. The class `Label` is an enumeration that contains the three different values that can be associated with these transitions.

- `ReachabilityRelation`. It contains a map which specifies for each outgoing transition of the sub-property the set of reachable incoming transitions of the same sub-property. It is bases on a set of reachability entries described in the `ReachabilityEnty` class.

- `Constraint` contains a set of sub-properties (at most one for each black box state) that specifies the set of the properties the developer must satisfy in the refinement process.

6.4.1 Constraint Input/Output Module

The `CHIAConstraintIO` module contains the classes which are used to load and save the constraint and the replacement from and to XML files. Figure 6.6 contains the class diagram associated with the `Constraint` class of the `CHIAConstraint` module. The same classes with different implementations are associated with the replacement.

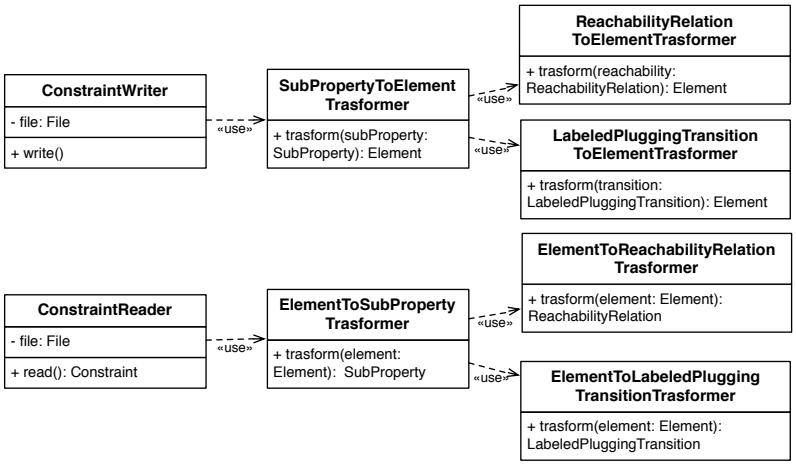


Figure 6.6: *The class diagram of the `CHIAConstraintsIO` module.*

- `ConstraintWriter`. It writes the constraint to an appropriate file.
- `SubPropertyToElementTrasformer`. It transforms the sub-property into the corresponding `XML` element.
- `ReachabilityRelationToElementTrasformer`. It converts the reachability relation between the incoming and the outgoing transitions of the sub-property into the corresponding `XML` elements.
- `LabeledPluggingTransitionToElementTransformer`. It converts an incoming or outgoing transition of the sub-property into the corresponding XML element.
- `ConstraintReader`. Given an `XML` input file, it loads the constraint from the file.
- `ElementToSubPropertyTrasformer`. It converts an `XML` element into the corresponding sub-property.
- `ElementToReachabilityRelationTrasformer`. It transforms an `XML` element into the corresponding reachability relation.
- `ElementToLabeledPluggingTransitionTrasformer`. It converts an `XML` element into the corresponding incoming or outgoing transition.

As previously mentioned, the `ConstraintReader` is used to load the constraint from an `XML` file. The `XML` files must satisfy the conditions specified in the `XSD` file

`Constraint.xsd`. Listing 6.2 presents a portion of the `XML` file of one of the constraints generated by the model checking algorithm in the motivating example used along the Chapter 5.

```
<constraint>
    <subproperty indispensable="true" modelstateId="3" name="send2">
        <ba>
            <propositions>
                <proposition name="send"/>
                <proposition name="success"/>
            </propositions>
            <states>
                <state id="11" name="3 − 2 − 0"/>
                <state id="14" name="3 − 1 − 1"/>
            </states>
            <transitions>
            <transition destination="11" id="15" propositions="SIGMA" source="11"/>
                <transition destination="14" id="22" propositions="send^!success" source="11"/>
                <transition destination="14" id="21" propositions="!success" source="14"/>
            </transitions>
        </ba>
        <intransitions>
            <plugtransition label="Y" id="1">
                <sourcestate>
                    <state id="2" name="send1"/>
                </sourcestate>
                <destinationstate>
                    <state id="11" name="3 − 2 − 0"/>
                </destinationstate>
                <trans id="23" propositions="fail"/>
            </plugtransition>
          ....
        </intransitions>
        <outtransitions>
            <plugtransition label="R" id="5">
                <sourcestate>
                    <state id="14" name="3 − 1 − 1"/>
                </sourcestate>
                <destinationstate>
                    <state id="4" name="q2"/>
                </destinationstate>
                <trans id="20" propositions="fail"/>
            </plugtransition>
        </outtransitions>
        <lowerReachability>
            <reachabilityElements/>
        </lowerReachability>
        <upperReachability>
            <reachabilityElements/>
        </upperReachability>
    </subproperty>
    <subproperty indispensable="true" modelstateId="2" name="send1">
        ....
    </subproperty>
</constraint>
```

Listing 6.2: *A portion of the constraint generated by the motivating example used along the Chapter 5.*

The constraint is made by two sub-properties which refer to the box states $send_1$ and $send_2$, respectively. Each sub-property contains a BA (which specifies the property the developer must satisfy in the refinement activity), the incoming and outgoing transitions, which describe how the claim is related to the system under development, and the reachability relations.

The `ReplacementReader` class allows loading the replacement of a box from an `XML` file. The `XML` files must satisfy the specification contained

in the XSD file Replacement.xsd. The ElementToReplacement transformer converts an XML element into the corresponding JAVA object using the ElementToTransition transformers and the other transformers previously described, such as the ElementToIBA transformer.

```
<replacement modelstateId="2" name="send1">
    <iba>
      <propositions>
        ...
      </propositions>
      <states>
        <state id="4" name="q4" />
        ...
        <state accepting="true" id="6" name="q6" />
      </states>
      <transitions>
        <transition destination="5" id="8" propositions="send" source="4" />
        <transition destination="6" id="9" propositions="wait" source="5" />
      </transitions>
    </iba>
    <intransitions>
      <plugtransition id="1">
        <sourcestate>
          <state id="1" name="q1" />
        </sourcestate>
        <destinationstate>
          <state id="4" name="q4" />
        </destinationstate>
        <trans id="1" propositions="start" />
      </plugtransition>
    </intransitions>
    <outtransitions>
      <plugtransition id="2">
        <sourcestate>
          <state id="6" name="q6" />
        </sourcestate>
        <destinationstate>
          <state id="3" name="send2" />
        </destinationstate>
        <trans id="2" propositions="fail" />
      </plugtransition>
      ...
    </outtransitions>
</replacement>
```

Listing 6.3: *A portion of the XML corresponding to the replacement described in Figure 5.3b.*

Listing 6.3 presents a portion of the XML file of the replacement corresponding to the replacement described in Figure 5.3b. The XML file contains the IBA which refines the box and the incoming/outgoing transitions which specify how the replacement is plugged into the original model.

6.5 Constraint computation

The CHIAContraintComputation module is used when the property is *possibly* satisfied to compute the sub-property associated to each black box state. The class diagram of the module is represented in Figure 6.7.

- ConstraintGenerator. It is the class that contains the entry point for the constraint computation. Given the model, the property and the intersection automaton computed by the CHIA checker, the ConstraintGenerator class

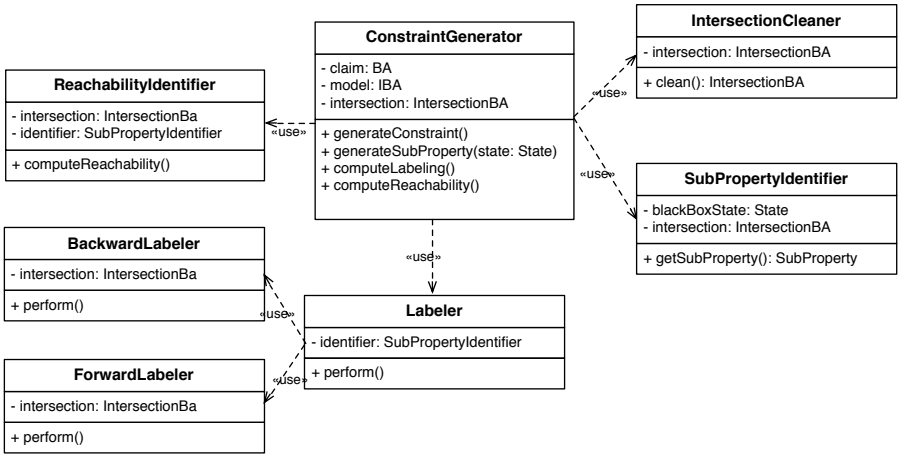


Figure 6.7: *The class diagram of the* `CHIAContraintComputation` *module.*

computes the corresponding constraint. The computation of the label associated to each incoming and outgoing transition and of the reachability relation are implemented as features that can be conveniently activated and deactivated.

- `IntersectionCleaner`. It implements the procedure described in Section 5.2.1, i.e., it removes from the intersection automaton the states from which it is not possible to reach an accepting state that can be entered infinitely many often, since these states are not useful in the constraint computation.

- `SubPropertiesIdentifier`. It extracts the sub-properties from the intersection automaton, by identifying the portions of the state space (the set of the mixed states and the transitions between them) that refer to the same box of the model.

- `Labeler`. It computes the values associated with the incoming and outgoing transitions of the sub-properties. It uses the two functions `BackwardLabeler` and `ForwardLabeler` which implement Algorithms 5 and 6.

- `ReachabilityIdentifier`. It computes for each sub-property the upper and the lower reachability relations as specified in Section 5.2.2.

6.6 Replacement checker

The `CHIAReplacementChecker` module contains the classes that allow checking whether the replacement of a black box state satisfies the corresponding constraint. Figure 6.8 presents the class diagram of the `CHIAReplacementChecker` module.

- `ReplacementChecker`. It is the entry point of the module. It takes as input the sub-property and a replacement for one of the black box states involved in the constraint. As the `Checker` class, it returns a value specified by the `SatisfactionValue` enumeration depending on whether the replacement satisfies, possibly satisfies or does not satisfy the property of interest.

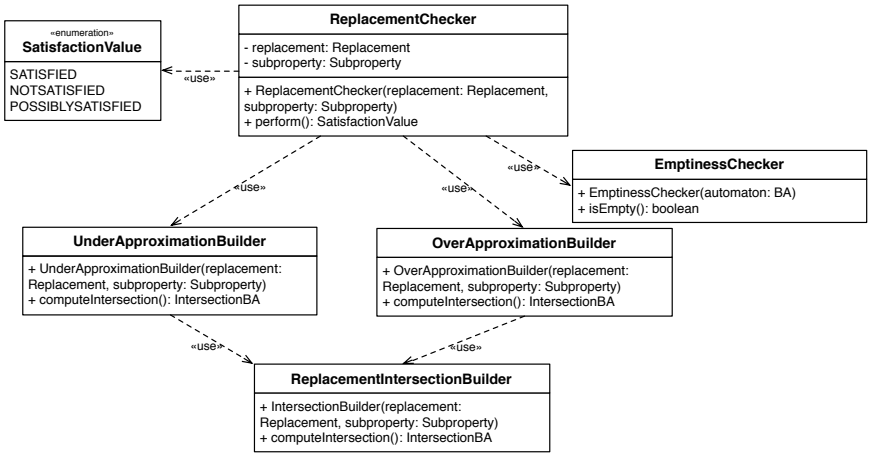


Figure 6.8: *The class diagram of the* `CHIARefinementChecker` *module.*

- `UnderApproximationBuilder`. It is used to compute the under approximation of the intersection automaton as specified in Section 5.3.1. It uses the `ReplacementIntersectionBuilder`.
- `OverApproximationBuilder`. It is used to compute the over approximation of the intersection automaton as specified in Section 5.3.1. It uses the `ReplacementIntersectionBuilder`.
- `ReplacementIntersectionBuilder`. It computes the intersection between the automaton associated with the replacement and the automaton of the corresponding sub-property. The intersection automaton is used by the `UnderApproximationBuilder` and the `OverApproximationBuilder` to compute the under and over approximation, respectively.

CHAPTER 7

Case Study

"Many of life's failures are people who did not realize how close they were to success when they gave up."

Thomas Edison, 1847-1931

We illustrate the applicability of the approach over two different case studies. For each case study we describe two different refinement scenarios. The first case study is a classical computer science example and concerns the well known mutual exclusion system, which has been considered in several works, such as [9, 104]. The second is a real case study which has been described in [139]. The case studies have been slightly modified since at the current stage our approach only supports *sequential* systems, i.e., it does not support parallel execution. Furthermore, some of the automata have also been manually converted from Kripke structures to Büchi automata using the procedure described in [33].

7.1 The mutual exclusion problem

The mutual exclusion problem considers two processes P_0 and P_1 which are competing for entering a critical section. Each of the two processes $P_i \mid i \in \{0, 1\}$ can be in three different states *a*) nt_i the process i is not in the critical section; *b*) tr_i the process i is in a waiting phase, i.e., the process i tries to enter its critical section but is waiting the permit from the controller; *c*) cr_i the process i is in its critical section. A semaphore is used to select the process that can enter the critical section. The semaphore can be in two states t_0 and t_1 meaning that the process P_0 and P_1 can enter into their critical section, respectively.

The following requirements are a set of well known requirements [9, 104] the designer must satisfy in the development of a mutual exclusion system. The first require-

ment, expressed through the *safety* Formula 7.1.1, specifies that two processes P_0 and P_1 must not be allowed to access simultaneously the critical section.

Requirement 7.1.1. *(safety)* $\phi_1 = G((\neg cr_0) \vee (\neg cr_1))$

The second requirement, described through the LTL *liveness* formula 7.1.2, specifies that starting from every state of the system, sooner or later each process will enter the critical section.

Requirement 7.1.2. *(liveness)* $\phi_2 = G(F(cr_0)) \wedge G(F(cr_1))$

The third requirement specifies that every waiting process will eventually enter its critical section.

Requirement 7.1.3. *(starvation freedom)* $\phi_3 = ((G(F(tr_0))) \rightarrow (G(F(cr_0)))) \wedge ((G(F(tr_1))) \rightarrow (G(F(cr_1))))$

Finally, the requirement 7.1.4 contains an unconditional fair condition which specifies that infinitely often finally at least one process is in its critical section.

Requirement 7.1.4. *(Unconditional fair condition)* $\phi_4 = G(F(cr_0 \vee cr_1))$

After these requirements have been defined the developer starts designing the mutual exclusion system. We consider two different refinement scenarios where the developer has a different feeling on the unknown parts of the system and how these unknowns are refined.

7.1.1 Scenario 1

When the developer designs the system he/she has to specify how the state of the system changes depending on the state of the two processes and the value of the semaphore. Instead of describing two separate cooperating processes, we describe their composition as one sequential process, since, up to now, our technique only supports sequential models.

The model described in Figure 7.1 describes a system that starts from the initial state q_1. The system randomly assigns the value t_0 or t_1 to the semaphore allowing the process P_0 or P_1 to enter its critical section and moving the system into the states q_2 or q_8, respectively. If the system is in the state q_2 and the process P_1 (P_0) tries to enter its critical section it moves to the state q_3 (q_4). If in the state q_3 (q_4) the process P_0 (P_1) tries to enter its critical section the state q_5 is reached. In the state q_5 both the processes have done a request to enter their critical sections and they are waiting for the response of the controller. Whenever the state q_5 or the state q_4 is reached the controller allows the process P_0 to enter the corresponding critical section reaching the states q_7 or q_6, respectively. Note that, when the system is in the state q_6 and the process P_0 is in its critical section, the process P_1 may also try to enter the critical section moving the system into the state q_7. When the system is in q_7, as soon as the process P_0 exits from its critical section, the value of the semaphore is changed into t_1 and the system moves into the box b_{10}. The developer does not specify the behavior of the system into the black box state b_{10}, he/she assumes it enters a component which is in charge of allowing the process P_1 to enter the corresponding critical section. When the system is in the state q_6 and the process P_0 exits its critical section, the controller changes the

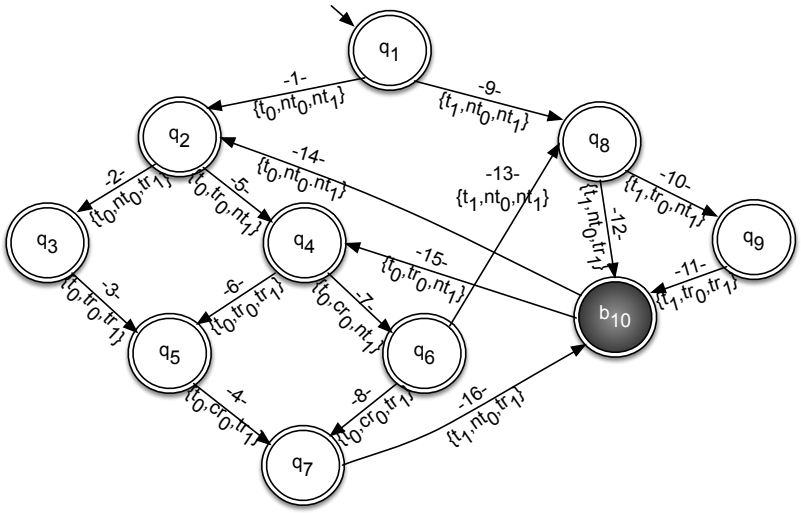


Figure 7.1: *The incomplete model the developer designs in the Scenario 1.*

value of the semaphore into t_1 and moves the system into the state q_8. The same state is reached from the initial state q_1 if the controller allows the process P_1 to enter its critical section, i.e., in the state q_1 the system randomly assigns an initial value to the semaphore. When the system is in the state q_8 and the process P_1 tries to access its critical section, the system moves to the box b_{10} which is in charge of allowing P_1 to access the corresponding critical section. If the system is in the state q_8 and the process P_0 tries to access its critical section it moves into the state q_9. If the system is in the state q_9 and the process P_1 tries to access its critical section it enters the replacement associated with the box b_{10}. The black box state b_{10} is left by two transitions performed when the system sets the semaphore to the value t_0 and the process P_1 leaves its critical section.

After the initial, high level model presented in Figure 7.1 is designed the developer wants to *check* whether the model satisfies, does not satisfy or possibly satisfies the requirements previously discussed. The `CHIA` framework offers the support for this activity. After the model and the properties associated to the different requirements are loaded the checking functionality allows the developer to answer his/her questions.

The *safety* requirement 7.1.1, which forces the two processes to not enter simultaneously their critical section, is possibly satisfied. Intuitively, the model of Figure 7.1 does not allow cr_0 and cr_1 to be true at the same time instant. However, there is no guarantee about the behavior of the system inside the box b_{10} which may yield a violation of the requirement.

The *liveness* requirement 7.1.2, which allows a process to finally enter its critical section, is also possibly satisfied. The model does not contain any violation behavior, i.e., it does not exists an accepting run in the intersection automaton which contains a state from which the critical section of P_0 and P_1 is not reachable any more. However, the property may be not satisfied inside the replacement of the box b_{10}, for example by simply providing a component that does not allow both the processes to enter their critical sections and never reach the outgoing transitions 14 and 15.

The *starvation freedom* requirement 7.1.3, specifying that a process asking to enter

its critical section has finally the permit to enter, is possibly satisfied. Imagine for example to consider the case in which the process P_1 is asking to enter its critical section when the system moves from the state q_2 to the state q_3. Since the value of the semaphore is equal to t_0, the system first allows the process P_0 to enter its critical section when it moves from the state q_5 to the state q_7. Then, the system enters the box q_{10} which must take care of allowing P_1 to enter the section cr_1. However, there is no guarantee that the replacement of q_{10} finally allows the process P_1 to enter the critical section.

Finally, also the *unconditional fair condition* specified in the requirement 7.1.4 is possibly satisfied. The unconditional fair condition specifies that infinitely often, finally, at least one process is in its critical section. This forces our system to have an infinite run and, starting from each state of the run, finally cr_0 or cr_1 is true. One of the possible ways to satisfy this condition is to connect an incoming transition of the box b_{10} to one of its outgoing transitions, allowing each run to finally satisfy the proposition cr_0, but still the satisfaction of the requirement depends on the replacement proposed for the box b_{10}.

To summarize, all the requirements are *possibly satisfied*, since no definitely accepting run exists in the model, and the behavior of the system over the possibly accepting runs depends on the replacement associated to the black box state b_{10}.

Whenever the property is possibly satisfied, the developer may be interested in knowing the constraint over the replacement of the box b_{10} that specifies the behaviors that do not violate the requirements of interest. `CHIA` supports the developer by computing a sub-property to be considered in the design of the replacement of the box b_{10}. The developer may consider the requirements one by one, iteratively generating the sub-properties associated to each LTL formula, or by computing a global sub-property obtained from the and combination of the requirements.

For the purpose of our discussion we imagine the developer is considering the unconditional fair condition specified in the Requirement 7.1.4. The constraint associated to this requirement is composed by a single sub-property which is represented in Figure 7.2. The sub-property specifies the possible ways in which the developer may violate the requirement 7.1.3. The incoming transition with source states q_7, q_8 and q_9 are marked as G since they are reachable in the intersection automaton through a run which contains only purely regular states, i.e., they are reachable independently of how the boxes are refined. The absence of outgoing transitions marked as R specifies that it is not possible to exit the replacement of the box and travel on a violating run made only by purely regular states. The absence of outgoing transitions marked as Y (both q_2 and q_4 are not marked) indicates that it is not possible to exit the replacement and travel on a possibly accepting run of the intersection automaton which does not involve states associated with the replacement of b_{10}. This is obvious by looking at the model presented in Figure 7.1. Whenever the box is left, the system reaches either the transition 7 or the transition 8 which guarantees that cr_0 is true and the property is satisfied. Thus, possible ways to violate the requirement imply the presence of accepting runs that guarantees both cr_0 and cr_1 to be not satisfied from a certain point inside the replacement. The reachability relation which involves the outgoing transitions 7 or the transition 8 specifies that it is possible to re-enter the replacement of b_{10}, i.e., the run that reaches the outgoing transition may not be violating by it-self but can be a prefix of a run that

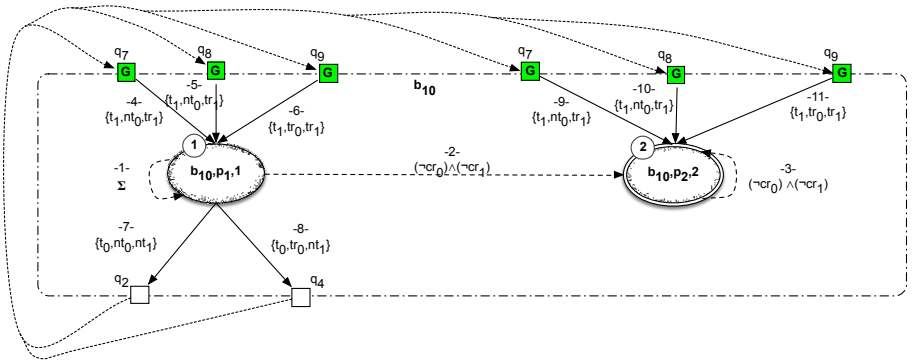


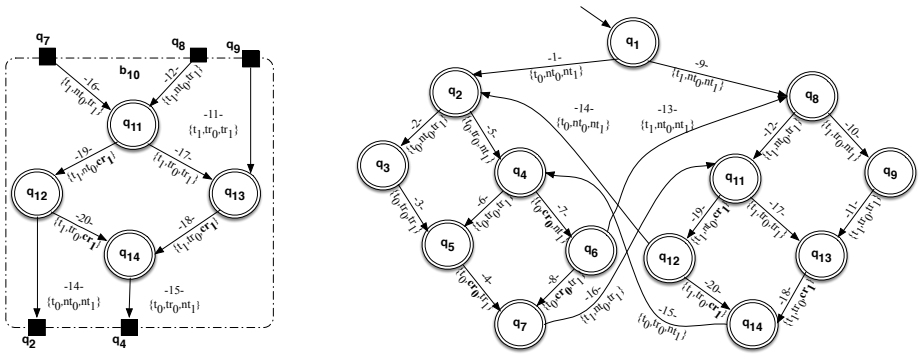
Figure 7.2: *The sub-property associated with the box b_{10} of the model presented in Figure 7.1 and the Requirement 7.1.3.*

violates the claim. The lower and upper reachability relation (described in Figure 7.2 through dotted arrows) specify how it is possible to reach the incoming transitions of b_{10} from its outgoing transitions through a run which involve only purely regular states or also involve mixed states that are not associated with the box b_{10}. For example, from the outgoing transition with destination q_2 it is possible to reach the incoming transition marked as G with source state q_7 (which in this case is also reachable from the initial state) through a run made by only purely regular states.

Starting from this sub-property, the developer may take two insights: *a*) by looking at the portion of the sub-property which refers to the state ② he/she may conclude that every suffix of an infinite run which does not contain a cr_0 or cr_1 is violating; *b*) every run which connect an incoming and outgoing transition of the replacement is violating if the previous condition is not satisfied.

After the sub-property described in Figure 7.2 has been computed, *the developer proposes a replacement for the box* b_{10}. For example, the developer may design the replacement described in Figure 7.3a. This replacement only contains finite runs, i.e., runs that enter/exit the replacement through its incoming/outgoing transitions.

The replacement guarantees that the proposition cr_1 is satisfied before the replace-



(a) *The replacement the designer proposes for the box b_{10}.*

(b) *The refinement $\mathcal{N}$ obtained when the replacement of the box b_{10} is plugged into the model $\mathcal{M}$.*

Figure 7.3: *The replacement and the refinement of the model presented in Figure 7.1.*

ment is left. More precisely, whenever the replacement is entered by the incoming transitions with source states q_7, q_8, the state q_{11} is reached. Note that when q_{11} is reached, the value of the semaphore is t_1 and the process P_1 is trying to enter its critical section. In this state, the controller may move the process P_1 into its critical section, by moving the system from the state q_{11} to the state q_{12}, or the process P_0 may also try to enter its critical section, making the system moving to the state q_{13}. This state is also reached from the incoming transition with source state q_9. When the system is in the state q_{12}, the refinement can be left whenever the process P_1 leaves its critical section, or it can move to the state q_{14} if the process P_0 tries to enter the critical section. The system moves from q_{13} to q_{14} when the controller allows the process P_1 to enter its critical section. Finally, the state q_{14} of the replacement is left through the outgoing transition that reaches the state q_4 whenever the process P_1 leaves its critical section.

The refinement obtained by plugging the replacement of the box b_{10} presented in Figure 7.3a in the model described in Figure 7.1 according to the Definition 4.2.8 is depicted in Figure 7.3b.

After the replacement presented in Figure 7.3a has been proposed, the developer may want to *check whether the refined version of the model satisfies the unconditional fair requirement*. One possibility is to generate the refinement of the model presented in Figure 7.3b and check it against the original requirements. However, the procedure described in Section 5.3 allows considering only the replacement against the previously generated constraint (i.e., the corresponding sub-property of the constraint). `CHIA` generates the upper and lower approximation of the intersection automaton as described in Section 5.3.2 and performs the emptiness checking procedures. Since in both cases the automata are empty, the property of interest is satisfied. The developer can use the replacement verification module of the `CHIA` model checking tool to verify the requirements 7.1.1, 7.1.2 and 7.1.2 against the previously generated constraints. The replacement satisfies all the requirements of interest.

7.1.2 Scenario 2

In the second scenario we consider a different developer in charge of designing the mutual exclusion system. As previously, we assume the developer does not specify the behavior of the two processes in isolation, but considers the behavior of the global system. However, the developer has a different notion about the domain he/she is trying to model and, in particular, he/she is more uncertain about the behavior of the system under development.

The developer may propose, for example, the model described in Figure 7.4. The system starts from the initial state q_1 and nondeterministically assigns an initial value to the semaphore. Depending on whether the process P_0 or P_1 is allowed entering its critical section, the system moves into the states b_{11} or q_8, respectively. The state b_{11} is a black box state, i.e., the developer still has to define the behavior of the system inside the box b_{11}. The developer specifies that it is possible to leave b_{11} either when the process P_1 tries to enter its critical section reaching the state q_3, when both the processes P_0 and P_1 try to enter their critical sections reaching the state q_5 or when the process P_0 is in its critical section. If the system is in the state q_3 and the process P_0 tries to enter its critical section it moves to the state q_5, i.e., in the state q_5 both the processes have done a request to enter their critical sections and they are waiting for

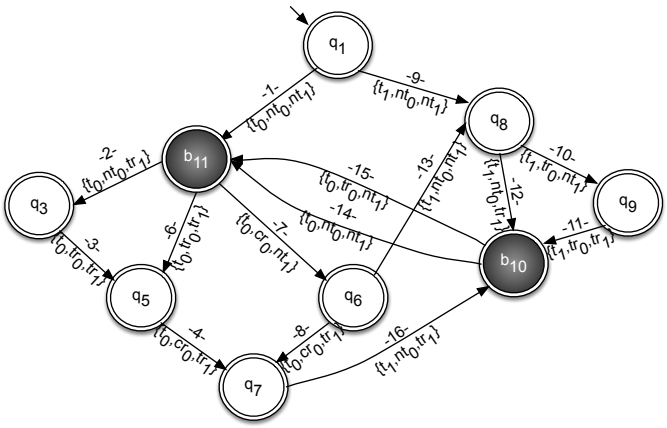


Figure 7.4: *The incomplete model the developer designs in the Scenario 2.*

the response of the controller. Whenever the state q_5 is reached the controller allows the process P_0 to enter the corresponding critical section reaching the state q_7. Note that, when the system is in the state q_6 and the process P_0 is in its critical section the process P_1 may also try to enter the critical section moving the system into the state q_7. When the system is in q_7, as soon as the process P_0 exits from its critical section, the value of the semaphore is changed into t_1 and the system moves into the black box state b_{10}. The developer does not specify the behavior of the system into the box b_{10}, he/she assumes it enters a component which is in charge of allowing the process P_1 to enter the corresponding critical section. When the system is in the state q_6 and the process P_0 exits its critical section, the controller changes the value of the semaphore into t_1 and moves the system into the state q_8. The same state is reached from the initial state q_1 if the controller allows the process P_1 to enter its critical section, i.e., in the state q_1 the system randomly assigns an initial value to the semaphore. When the system is in the state q_8 and the process P_1 tries to enter its critical section, the system moves into the black box state b_{10}, in charge of allowing P_1 to access the corresponding critical section. If the system is in the state q_8 and the process P_0 tries to access its critical section it moves into the state q_9. If the system is in the state q_9 and the process P_1 tries to enter its critical section it enters the replacement associated with the box b_{10}. The box b_{10} is left by the transitions performed when the system sets the semaphore to the value t_0 and the processes P_0 and P_1 are not trying to access the critical section, or when the semaphore has the t_0 value and the process P_0 and P_1 is trying and is not trying to enter the critical section, respectively.

As in the previous scenario, after the initial, high level model is designed, the developer wants to *check* whether the *incomplete* model satisfies, does not satisfy or possibly satisfy the safety, liveness, starvation freedom and the unconditional fair properties. The CHIA framework supports the developer in this activity. After the model and the properties of interest are loaded, the model checking functionality allows answering his/her questions.

The *safety* requirement 7.1.1, which forces the two processes to not enter simultaneously their critical section, is possibly satisfied. The model of Figure 7.4 does not allow cr_0 and cr_1 to be true at the same time instant, but there is no guarantee about

the behavior of the system inside the replacements of the boxes b_{10} and b_{11} which may cause a violation of the requirement.

The *liveness* requirement 7.1.2, which allows a process to finally enter its critical section, is also possibly satisfied. There is no violating behavior, i.e., it does not exists an accepting run in the intersection automaton which contains a state from which the critical section of P_0 and P_1 is not reachable any more. However, it is also possible to never reach the critical section of P_0 or P_1, e.g., by providing a replacement for b_{11} which does not allow to reach the outgoing transitions 2, 6 and 7 and never satisfy the propositions cr_0 or cr_1.

The *starvation freedom* requirement 7.1.3, which states that a process asking to enter its critical section has finally the permit to enter, is also possibly satisfied. For example, when the system moves from q_8 to b_{10}, P_1 tries to enter its critical section but there is no guarantee that inside the replacement of the black box b_{10} the process P_1 has the possibility to enter that section.

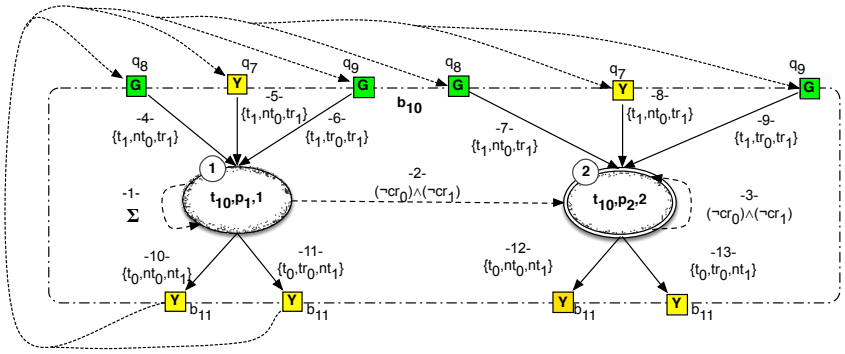
The *unconditional fair condition* specified in the requirement 7.1.4 is also possibly satisfied. The unconditional fair condition specifies that infinitely often, finally, at least one processes is in its critical section. Even if every infinite accepting run contains either the transition 4 or the transition 7, it may be the case in which b_{11} and b_{10} do not allow to reach one of its outgoing transitions and their replacements do not allow one of the processes to enter its critical section.

To summarize, as in Scenario 1, all the requirements are possibly satisfied. Indeed, no accepting run exists in the model that do not depend on the replacements associated to the black box states b_{10} and b_{11}. This is also consistent with Theorem 4.2.1 and the observation that the model proposed in the first scenario is a refinement of the model described in Figure 7.4.

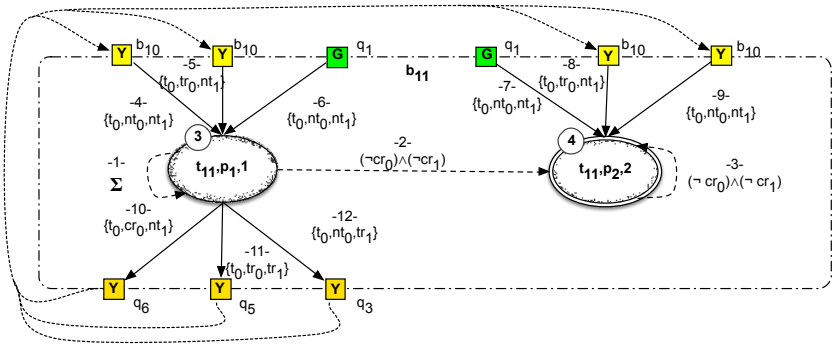
After the developer has checked its incomplete model against the properties of interest, he/she may be interested in some guidelines on how to design the replacements of the black box states b_{10} and b_{11} in a way that do not violate the requirements of interest. These behaviors are specified through the sub-properties associated to the boxes b_{10} and b_{11}. As in the previous scenario, the developer may consider the requirements one by one iteratively generating the sub-properties associated to each LTL formula or by computing the global sub-property associated to the conjunction of the sub-properties.

For the sake of clarity, in our discussion we imagine that the developer is considering the unconditional fair condition specified in the Requirement 7.1.4. The constraint associated to this requirement is composed by two sub-properties, one for each box of the model, that are presented in Figure 7.5a and 7.5b. The sub-properties specify the possible ways in which the developer may violate the requirement 7.1.3.

The sub-property associated with the black box state b_{10} specifies that a run which traverses the box b_{10} and reaches the state b_{11} is a possibly violating run indeed the box b_{11} may cause a violation of the property. Furthermore, every replacement that contains a run that at some point contains an infinite internal run in which $(\neg cr_0) \wedge (\neg cr_1)$ is not true violates the unconditional fair condition. The incoming transitions with source states q_8 and q_9 are marked as G since they are reachable from the initial state in the intersection automaton through a run which contains only purely regular states, i.e., they are reachable indipendently on how the boxes are refined. The incoming transitions with source state q_7 are marked as Y since to reach q_7 it is necessary to



(a) *The sub-property associated with the box b_{10} of the model presented in Figure 7.4.*



(b) *The sub-property associated with the box b_{11} of the model presented in Figure 7.4.*

Figure 7.5: *The sub-properties associated with the black box states b_{10} and b_{11} of the model presented in Figure 7.4.*

traverse a run which contains mixed states, i.e., the replacement associated with the box state b_{10}. The outgoing transitions 10, 11, 12 and 11 are marked as Y since it is possible to reach an accepting state through a run that involves the replacements associated to other boxes. In particular, the accepting state is possibly contained inside the replacement of the box b_{11}. The reachability relation specifies how it is possible to reach an incoming transition of the sub-property associated with the black box state b_{10} from one of the outgoing transitions. In particular, the upper reachability relation specified in Figure 7.5a with dotted lines states that it is possible to reach the incoming transitions 4, 5, 6, 7, 8 or 9 from the outgoing transitions 10 and 11.

The sub-property associated with the box b_{11} specifies that it is possible to reach the incoming transitions 6 and 7 through a run which contain only purely regular states while it is possible to reach the incoming transitions 4, 5, 8 and 9 through a run which traverses some mixed states, i.e., the one associated with the replacement of the box b_{10}. Furthermore, from the outgoing transitions 10, 11 and 12 it is possible to reach an accepting state through a run which involves some mixed state which is not associated to the box b_{11}, i.e., the mixed states associated to the replacement of the box b_{10}. From these outgoing transitions, the upper reachability relation specifies that it is possible to reach the incoming ports 4, 5, 8 and 9.

In both the sub-properties, the absence of red outgoing transitions indicate that it is not possible to exit the replacement of the box and travel on a violating run made only

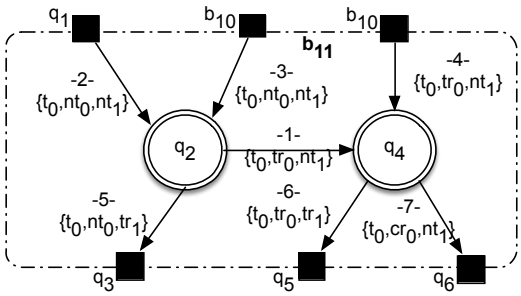


Figure 7.6: *The replacement designed for the box b_{11} of the model presented in Figure 7.4.*

by purely regular states, i.e., there must be the cooperation of one of the replacement (or both of them) to make the property (not) satisfied.

After the sub-properties presented in Figures 7.5a and 7.5b have been computed, the developer *proposes a replacement* for the box b_{11}. In the current scenario, we assume the developer proposing the replacement described in Figure 7.6, which generates the model associated with the first scenario described in Figure 7.1.

The replacement specifies that by firing the incoming transitions coming from b_{10} and q_1 and labeled with $t_0 \wedge tr_0 \wedge nt_1$, $t_0 \wedge nt_0 \wedge nt_1$ and $t_0 \wedge nt_0 \wedge nt_1$ it is possible to reach the states q_4 and q_2. The state q_2 specifies the condition in which the processes P_0 and P_1 are not trying to enter their critical sections, while the state q_4 is reached when the process P_0 is trying to enter its critical section and the process P_1 is not inside and neither trying to enter the corresponding critical section. The replacement is left through the outgoing transitions 5, 6 and 7 when the process P_1 tries to enter its critical section, when both the processes try to enter their critical sections, or when the system gives the permit to the process P_0 to enter its critical section, respectively.

After the replacement presented in Figure 7.6 has been proposed the developer may want to *check the replacement against the previously generated constraint*. As previously mentined, `CHIA` directly checks the replacement against the sub-property specified in Figure 7.5b, by generating the upper and lower approximation of the intersection automaton as described in Section 5.3.2 and performing the emptiness checking procedures. In this case, there is no violating behavior, i.e., the lower approximation of the intersection automaton is empty, but the upper approximation contains a possibly violating behavior. Indeed, by connecting the replacement to the states q_6, q_5 and q_3 the developer allows the system to reach the state b_{10} which can be refined in a way that makes its sub-property (and thus the original property) not satisfied. Thus, the property is possibly satisfied in the refined automaton.

7.2 The Pick and Place Unit

The effectiveness of the approach has been evaluated on a real case study presented in [139], where the evolution of a Pick and Place Unit (PPU) is analyzed[1]. The Pick and Place Unit is an open case study for analyzing evolution of automation systems,

[1] A video which describes the PPU unit in action can be found at `https://www.ais.mw.tum.de/research/equipment/ppu/`

since it is a limited size and complexity example, but it provides a valuable trade-off between complexity and evaluation effort [79]. The goal of the PPU unit is to move pieces (WP) to different locations of the production line. An overview of the PPU unit considered in this work is presented in Figure 7.7[2]. Four different components are considered along the different refinement scenarios: stack, crane, stamp and conveyor.

- The *stack* is the input storage of work pieces, where they are kept until the crane is able to move them (marked with ①).
- The *conveyor* is used as an output storage of work pieces (identified with ②).
- The *stamp* is used to stamp the work pieces (marked with ③).
- The *crane* is the transportation unit that moves pieces between the stack, the conveyor and the stamp (identified with ④).

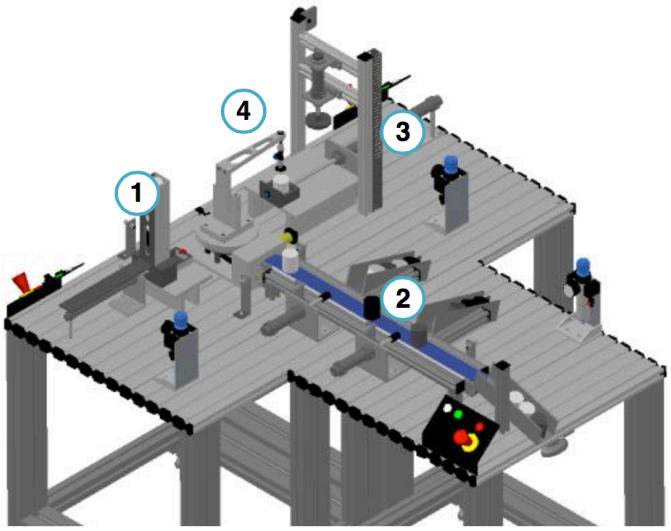


Figure 7.7: *A high level description of the PPU unit.*

Several evolution steps have been considered in [139]. For the purpose of our discussion we have transformed the original automata presented in [139] to fit the formalism proposed in this work. Furthermore, since our approach only supports sequential systems, the refinement scenarios have been modified.

We assume that the system to be developed is based upon the following propositions, which predicate over the occurrence of certain events into the system. For example, the proposition $stDIWPAV$ specifies an input event (DI) which states that a WP is available (AV) at the stack (st). The list of all the propositions which refers to the the stack, the stamp, the crane and the conveyor are presented in Table 7.1, 7.2, 7.3 and 7.4, respectively.

[2]The images of the PPU case study presented in this thesis have been taken from [139].

Stack	
Proposition	*Meaning*
$stDIWPAV$	a working piece (WP) is available at the stack for being piked up.
$stDIAS$	the crane has reached the stack.
$stDIEXT$	the pneumatic cylinder is extended.
$stEMPTY$	the stack is empty.
$stWPAV$	at least a working piece is on the stack.
$stDIEXT$	the separating cylinder has been extended.
$stACTEXT$	the pneumatic cylinder pushes, by extruding, the bottom WP from the stack.
$stDISEXT$	the separating cylinder of the stack is extended.

Table 7.1: *The atomic propositions of the* `stack` *component.*

Stamp	
Proposition	*Meaning*
$smDIWPAV$	a WP is available at the stamp for being printed.
$smDISTMP$	a working piece has been printed by the stamp.
$smDIWPSPR$	a printed WP is available at the stamp for being taken to the conveyor.
$smDITMP$	a timer event has been occurred.
$smDISTEXT$	the stamping cylinder has been extended.
$smDISTRET$	the stamping cylinder has been retracted.
$smDISLEXT$	the sliding cylinder has been extended.
$smDISLRET$	the sliding cylinder has been retracted.
$smACTSPW$	setting the pressure value.
$smACTSTEXT$	the stamp extends the stamping cylinder.
$smACTSTRET$	the stamp retracts the stamping cylinder.
$smACTSLEXT$	extending the sliding cylinder.
$smACTSLRET$	retracting the sliding cylinder.

Table 7.2: *The atomic propositions of the* `stamp` *component.*

Crane	
Proposition	*Meaning*
$crDIWPM$	the crane detects an available WP.
$crDIPUWP$	the piece has been picked up.
$crDIPCEXT$	pneumatic cylinder extended.
$crDIPCRET$	pneumatic cylinder retracted.
$crDITTTC$	the crane has reached the conveyor.
$crDIATSTRMP$	the crane has reached the stamp.
$crDITTTS$	the crane has reached the stack.
$crDIWPP$	plastic WP detected.
$crDIWPM$	metallic WP detected.
$crDIWPB - PW$	black (PB) or white (PW) WP detected.
$crACTPCRET$	retracting the pneumatic cylinder.
$crACTPCEXT$	extending the pneumatic cylinder.
$crACTPDWP$	opening the vaccum gripper.
$crACTPUWP$	closing the vaccum gripper.
$crACTTTSTMP$	moving the crane to the stamp.
$crACTTTS$	moving the crane to the stack.
$crACTTTC$	moving the crane the the conveyor.

Table 7.3: *The atomic propositions of the* `conveyor` *component.*

Conveyor	
Proposition	*Meaning*
$coDIWPAB$	a working piece is at the beginning of the conveyor.
$coDIWPRM$	the working piece has been released on the ramp.
$coDIWPRP1$	working piece at ramp one.
$coDIST1NFULL$	the first ramp is not full.
$coDIST2NFULL$	the secon ramp is not full.
$coDIWPRP1$	working piece detected at ramp one.
$coDIWPRP2$	working piece detected at ramp two.

$coDIT1TOUT$	conveyor timer 1 timeout.
$coDIT2TOUT$	conveyor timer 2 timeout.
$coDIT3TOUT$	conveyor timer 3 timeout.
$coDIC1EXT$	the first pushing cylinder has been extended.
$coDIC1RET$	the first pushing cylinder has been retracted.
$cDIPR1$	the piece has been placed on the first ramp.
$cDIPR2$	the piece has been placed on the second ramp.
$cDIPR3$	the piece has been placed on the third ramp.
$coACTTF$	turn forwards the conveyor.
$coACTC1EXT$	extending the first pushing cylinder.
$coACTC2EXT$	extending the second pushing cylinder.
$coACTMS$	stops the motor of the conveyor.

Table 7.4: *The atomic propositions of the* `conveyor` *component.*

7.2.1 Requirement analysis

The expected behavior of the `PPU` unit is described by the following requirements.

The working pieces are picked up from the stack and finally released on the conveyor. This can be formalized in Requirement 7.2.1 as specified by the formula ϕ_1.

Requirement 7.2.1. *(Every `WP` finally reaches the conveyor)*
$\phi_1 = G(stDIWPAV \rightarrow F(coDIWPAB))$

The requirement specifies that after the system signals that a piece is available at the stack (st) for being picked up ($stDIWPAV$), the system will finally signals that the piece has reached the conveyor ($coDIWPAB$)[3].

We have two types of `WP` metallic or plastic. If a piece is a metallic `WP` it must be stamped. This requirement is formalized trough the LTL formula ϕ_2 specified in Requirement 7.2.2.

Requirement 7.2.2. *(The metallic working pieces must be stamped)*
$\phi_2 = G((crDIWPM) \rightarrow (F(smDISTMP)))$

Formula ϕ_2 specifies that if the proposition $crDIWPM$, which specifies that a metallic working piece has been detected by the crane, is true at some point, then, finally, the proposition $smDISTMP$ must be true. The proposition $smDISTMP$ specifies that a piece has been printed.

After a piece has been stamped it must be released on the conveyor. This requirement is formalized trough the LTL formula ϕ_3 specified in Requirement 7.2.3.

Requirement 7.2.3. *(A stamped piece is finally released on the conveyor)*
$\phi_3 = G((stDISTMP) \rightarrow (F(coDIWPAB)))$

Formula ϕ_3 specifies that if a work piece has been printed, the atomic proposition $stDISTMP$ is true, it is finally released on the conveyor, i.e., the atomic proposition $coDIWPAB$ is true.

Finally, before preparing a new working piece it is necessary that the previous one has been released on the conveyor. This is specified by the Requirement 7.2.4.

Requirement 7.2.4. *(Before preparing a new piece the previous one must be released on the ramp)*
$\phi_4 = G((stDIWPAV) \rightarrow (\neg((\neg(stDIWPAV))U(coDIWPRM)))$

[3]Note that the piece that reaches the conveyor can be different than the original one.

Formula ϕ_4 specifies that after a piece is available ($stDIWPAV$), it is not true that the system prepares another piece ($stDIWPAV$) and the event $coDIWPRM$ which specifies that the WP is released on the ramp has not occurred before.

7.2.2 The PPU components

The developer starts by designing a high level description of the PPU unit. The initial high level design is presented in Figure 7.8. The system can be in four different control states, which are represented as black box states since the behavior of the PPU in these states still has to be defined. Whenever the system is in the stack state, the stack provides a working piece. The box is left by firing the transition 1 labeled with $stDIWPAV$ which specifies that a WP is available. The crane box contains the logic

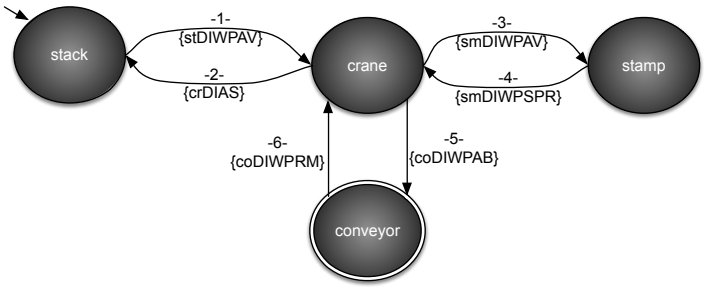


Figure 7.8: *The PPU overall architecture.*

to move the pieces between the stack, the stamp and the conveyor. Whenever the crane reaches the stack ($crDIAS$ is true) the system enters the replacement associated to black box state that represents the stack which contains the logic to provide a piece. After the crane provides a piece to the stamp device, the transition 3 is performed. The transition is fired when the proposition $smDIWPAV$ is true, which specifies that a WP is available at the stamp. The stamp component is left whenever the transition 4 is performed, i.e., the piece is ready to be moved to the conveyor ($smDIWPPR$). Whenever a component is ready at the beginning of the conveyor ($coDIWPAB$), the transition 5 is fired and the conveyor black box accepting state is reached. The box is left whenever the piece has placed in one of the ramps ($coDIWPRM$).

We imagine the developer is in charge of designing the behavior of the crane, while the stack, stamp and conveyor behaviors are designed by third parties as described in the following.

The *stack* represents an input storage for work pieces. It contains three different parts: a magazine of pieces (indicated as ① in Figure 7.9) where the work pieces are stacked, a pneumatic cylinder (indicated as ② in Figure 7.9) to get the lower most piece and the micro-switch (indicated as ③ in Figure 7.9) that indicates whether there exists a piece in the pick-up position. The pneumatic cylinder is equipped with two binary sensors indicating the position of the cylinder, i.e., if it is extended or retracted.

The behavior of the stack component is described in Figure 7.10. The stack is started from its initial state q_1 or when the crane reaches the stack ($stDIAS$ is true), which means the system is waiting for a new working piece. If the stack is empty, the system moves to the state q_3, otherwise a WP is available and the system moves to q_2. The

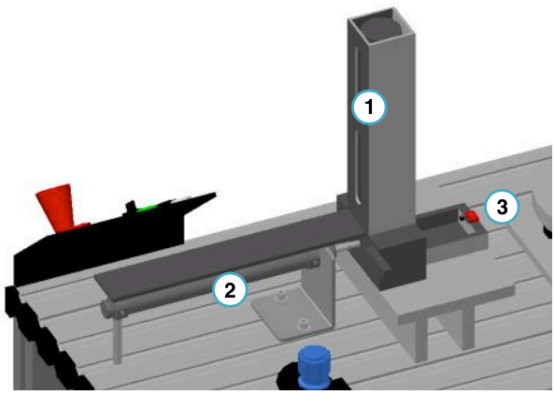


Figure 7.9: *The* `Stack` *component structure.*

system moves from the state q_3 to q_2 as soon as a new piece is available. In the state q_2, the system activates the pneumatic cylinder of the stack acting as separator which pushes, by extruding, the bottom work piece from the stack ($stACTEXT$). As soon as the piece is available ($stDIEXT$), the system fires the transition 12 and the system moves to the state q_4. In order to ensure the correct position of the `WP`, the separating cylinder of the stack is extended by firing the transition 13 ($stACTSEXT$). As soon as this process is completed, transition 14 is fired and the replacement is left through the transition 1.

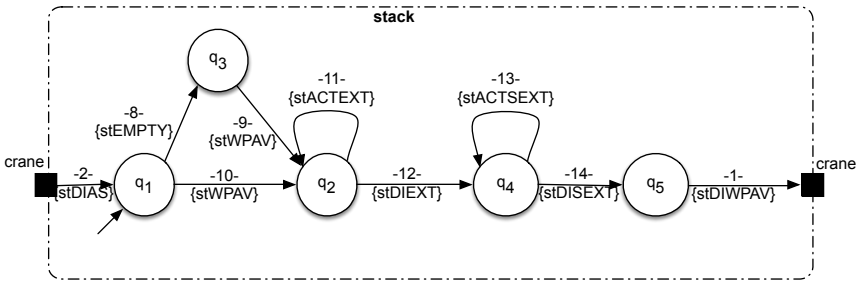


Figure 7.10: *The* `Stack` *component behavior.*

The *stamp* module, whose structure is described in Figure 7.11, is used to stamp work pieces and is made by five different components: *a*) the *sliding cylinder* is used to extend and retract work pieces into or from the stamp component (indicated as ①). The sliding cylinder consist of a valve to extend and retract the cylinder as well as two binary end position sensors for detecting whether the cylinder is extended or retracted; *b*) the *stamping cylinder* is used to stamp the work piece (indicated as ②). The end position sensors are used to indicate whether the stamping cylinder is lowered or raised; *c*) the *micro switch* detects whether the `WP` has been placed by the crane on the stamp component (indicated as ③); *d*) the *pressure sensor* detects the current pressure on the `WP` (indicated as ④); *e*) the *proportional valve* is used to set a specific pressure on the component (indicated as ⑤).

The behavior of the stamp component is described in Figure 7.12. Whenever the stamp detects a `WP` ($smDIWPAV$) the state q_6 is reached. The stamps activates the

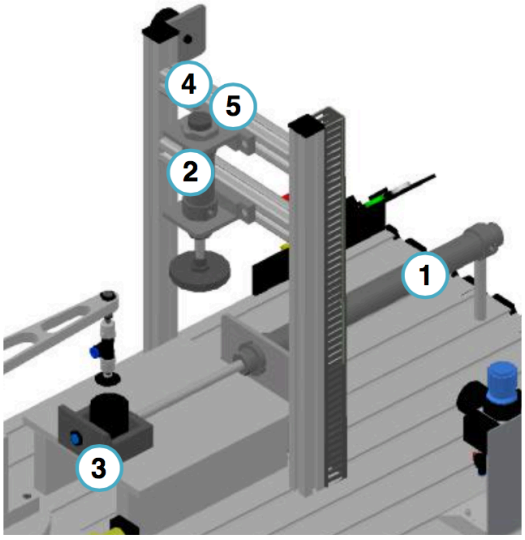


Figure 7.11: *The* `Stamp` *component structure.*

sliding cylinder ($smACTSLRET$). As soon as the sliding cylinder has been retracted ($smDISLRET$), the system moves in q_7 and the pressure value is set through the transition 17. The system extends the stamping cylinder ($smACTSTEXT$) through transition 18. Whenever the extension phase is finished, i.e., the transition 19 is performed, and the state q_9 is reached. In the state q_9 the system is printing the working piece. The state q_9 is left as soon as a timer event occurs, i.e., the system fires the transition 20 which specifies that a piece has been stamped ($smDISTMP$). The stamp component subsequently retracts the stamping and the sliding cylinder by performing the transitions 21, 22 and 23, 24, respectively. Finally, the transition 4 which specifies that the piece is ready to be moved to the conveyor ($smDIWPSPR$) is fired.

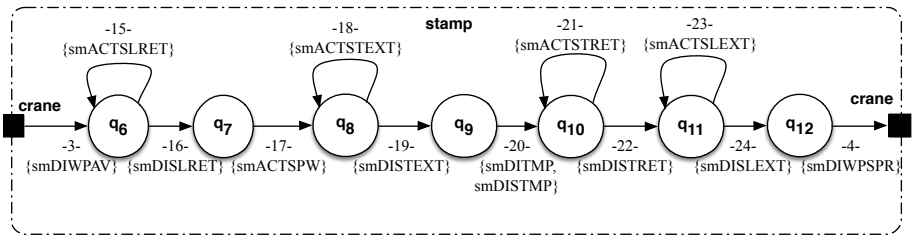


Figure 7.12: *The* `Stamp` *component behavior.*

The *conveyor* is a transportation component that takes the work pieces in three different ramps which are used as output storage. The structure of the conveyor component is described in Figure 7.13. Two of the ramps are filled using the respective pushing cylinders. The third ramp is mounted at the end of the conveyor and is filled by moving the work piece using the conveyor. The conveyor contains the following components: *a*) the *motor* for realizing the translational movement of work pieces (indicated as ①); *b*) the *presence sensor* which is mounted at the beginning and at the end of the conveyor to detect whether a work piece is placed on the conveyor (indicated as ②); *c*) the *three ramps* which are filled with the working pieces; *d*) the two *pushing cylinders* installed

to push work pieces in the first two ramps (indicated as ③); *e*) the *optical sensor* it is used to detect the presence of a piece at a particular position of the conveyor (indicated as ④); *f*) the *valves* are used for extending and retracting the cylinders; *g*) the *end position sensors* are used for detecting whether the cylinders have been extended or retracted.

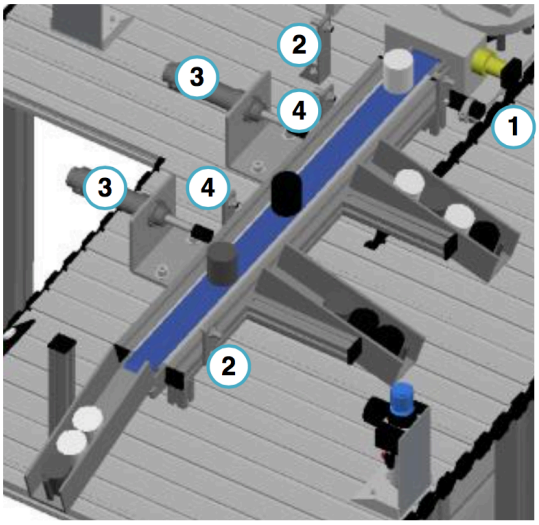


Figure 7.13: *The* `Conveyor` *component structure.*

The conveyor behavior is described by the automaton depicted in Figure 7.14. Once the `WP` is placed on the conveyor, it is detected by a sensor and the state q_{13} is reached and the actuator which turns forwards the conveyor is activated (transition 25). Three types of behaviors can be exhibited by the conveyor: *a*) as soon as the `WP` reaches the optical sensor positioned at the beginning of the first ramp, if the ramp is not full the transition 26 fires and the system goes into the state q_{14}. The system waits a time that depends on the conveyor belt speed and the distance between the sensor and the pushing cylinder. As soon as the timeout events occurs, the transition 29 is fired. The cylinder c_1 of the conveyor is subsequently extended (transition 30) and retracted (transition 32). Whenever the cylinder is finally retracted the state q_{17} is reached; *b*) if the `WP` is not placed in the first ramp, the first pushing cylinder remains in the retracted position and the work piece passes it. Next the work piece passes the second optical sensor, locate before the second ramp and, if the ramp is not full, the transition 27 is fired and the state q_{19} is reached. As in the previous case, the system waits a time that depends on the conveyor belt speed and the distance between the sensor and the pushing cylinder. As soon as the timeout events occurs, the transition 35 is fired. The cylinder c_2 of the conveyor is subsequently extended (transition 36) and retracted (transition 38). Whenever the cylinder is finally retracted the state q_{17} is reached; *c*) if instead the `WP` is not placed in the second ramp, the transition 28 is fired and the system moves to the state q_{22}. The system goes into the state q_{17} when a timeout event which implies that the `WP` has reached the ramp occurs. As soon as the state q_{17} is reached the motor is stopped by firing the transition 34 labeled with $coACTMS$ and the conveyor component is left.

Finally, the *crane* is a transportation unit that moves pieces between the stack, the

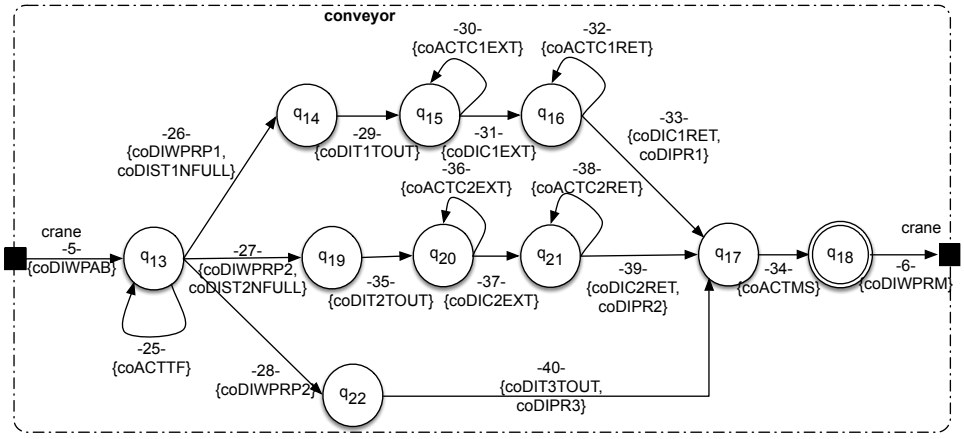


Figure 7.14: *The* `Conveyor` *component behavior.*

conveyor and the stamp. The crane structure is represented in Figure 7.15 and it is composed by the following components: *a*) a *pneumatic cylinder* for lifting and lowering work pieces (indicated as ①). The cylinder posses two binary position sensors, one at each end to detect whether the cylinder is extended or retracted as well as the valve as actuator. *b*) a *vacuum gripper* controlled by two valves which is used to pick and release pieces (indicated as ②). A *micro switch* indicates whether a work piece is gripped. *c*) a *turning table* which allows rotational movement (indicated as ③). *d*) a set of *digital sensors* are installed at the bottom of the turning table (indicated as ④). These sensors indicate whether the stack, the stamp and the conveyor is reached. *e*) a *motor* allows the rotational movement of the turning table (indicated as ⑤).

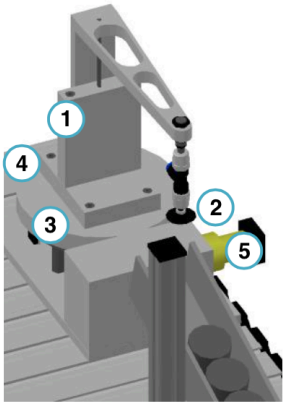


Figure 7.15: *The* `Crane` *component structure.*

7.2.3 Scenario 1

In the first refinement scenario the developer is in charge of designing the behavior of the crane given the components previously described. Before performing the refinement activity he/she *checks* whether the requirements 7.2.1, 7.2.2,7.2.3 and 7.2.4 are satisfied, not satisfied or possibly satisfied. The requirement 7.2.1 is satisfied by the proposed design. The state q_{18} is the only accepting state in the model since the `crane`

state is not accepting and as a consequence cannot contain any accepting state. Since the automata based model checking framework considers only accepting run, i.e., runs that enters the state q_{18} infinitely many often, the runs have to traverse the transition 5 which is labeled with $coDIWPAB$ and imply the satisfaction of the property. The requirement 7.2.2 is possibly satisfied. Intuitively the developer may propose a replacement that does not take the metallic work pieces to the stamp. The requirement 7.2.3 is satisfied, i.e, every piece that has been printed is finally released on the conveyor. Indeed, as previously, the accepting runs must traverse the state q_{18} which can be reached only by firing the transition 5 which require the `WP` to be released on the conveyor. Finally, the requirement 7.2.4 is possibly satisfied, the developer may design a replacement that release a type of piece on the conveyor and another piece is left on the stamp.

Before proceeding in the design of a new replacement, the developer *computes the sub-properties* the crane must satisfy to guarantee the system possesses the properties of interest. The sub-property associated with the requirement 7.2.2 specifies that the only possible way to violate the property is to provide a run that enters infinitely many times in the conveyor and the metallic piece is detected but it is not taken to the stamp. The sub-property associated with the requirement 7.2.4 specifies all the possible ways in which the requirement can be violated. There are many ways to violate this requirement. For example, a violating behavior is a run that enters infinitely often the conveyor but non deterministically, it can fire the transition 2 before reaching the conveyor. Another possibility is that whenever a piece is taken to the stamp it is not collected. The sub-property encodes all these behaviors and how they are related with the `PPU` overall architecture depicted in Figure 7.8.

The *replacement proposed* by the developer for the black box state representing the crane is depicted in Figure 7.16. As soon as a piece is available on the stack to be picked up, i.e., the transition 1 is fired,and the crane turns to the stack (transition 41). As soon as the crane reaches the stack, transition 42 is fired, and the pneumatic cylinder is extended (transitions 43 and 44). The vacuum gripper is closed to pick up the piece (transition 45). As soon as the piece has been picked up, transition 46 is fired and the pneumatic cylinder is extended (transitions 47 and 48). Depending on whether the piece is a metallic (transition 55) or plastic (transition 49) piece the state q_{31} or q_{28} is reached. If the system reaches the state q_{28} the turning table is rotated to the conveyor (transitions 50 and 51), the pneumatic cylinder is extended (transition 52 and 53) and the vacuum gripper is open (transition 54). As soon as the procedure is finished the conveyor is notified that a new `WP` is available (transition 5). If instead the piece is metallic, the crane is moved to the stamp (transitions 56 and 57) the pneumatic cylinder is extended (transition 58 and 59) and the vacuum gripper is open (transition 60). As soon as the procedure is finished the stamp is notified that a new `WP` is available for being printed (transition 3). When a printed piece is available on the stamp, transition 4 is fired. First, the crane is moved to the stamp (transitions 61 and 62), the pneumatic cylinder is extended (transitions 63 and 64) and the piece is picked up (transitions 65 and 66). Then, the cylinder is retracted (transitions 67 and 68), the crane is moved to the conveyor (transitions 69 and 70), the cylinder is extended (transitions 71 and 72) and the vacuum gripper is opened (transition 73). Finally (transition 5), the conveyor is notified that a new `WP` is available. If the conveyor notifies the crane by firing the transition 6 that the `WP` has been placed on one of the output stacks, the crane moves to

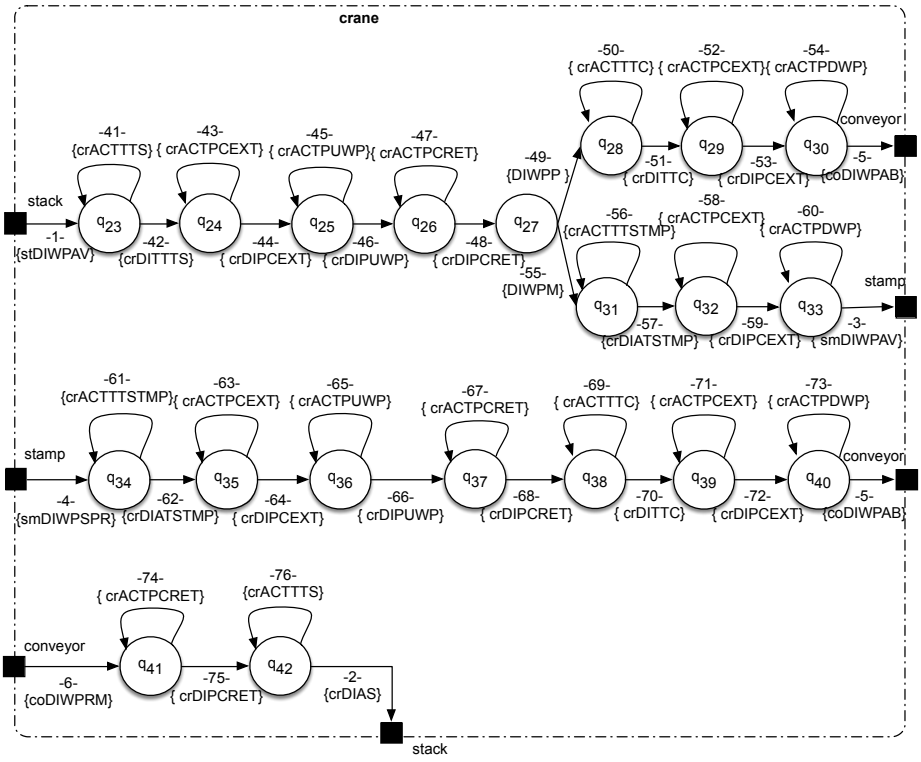


Figure 7.16: *The* `Crane` *behavior proposed in the first refinement Scenario.*

the input stack where it waits that a new `WP` is available. First, the pneumatic cylinder is retracted (transitions 74 and 75), then the turning table is moved to the stack (transitions 76). As soon as the stack is reached, the transition 2 is fired and the control is given to the stack component.

After the replacement has been designed, the developer wants to *check* whether the properties of interest are satisfied, possibly satisfied or not satisfied by the refined automaton. `CHIA` allows checking the replacement against the previously generated constraints. The tool notifies the developer that all the properties of interest are satisfied by the replacement. Thus, the replacement can be added to the system.

7.2.4 Scenario 2

In the second refinement scenario, the developer re-design the behavior of the `crane` component. Since he/she already performed the *checking* of the incomplete model, he/she knows that the requirements 7.2.1 and 7.2.3 are already satisfied, while the requirements 7.2.2,7.2.4 must be satisfied by his/her replacement.

However, in this *refinement round* the developer decided to add an additional check (done with a corresponding optical sensor) which allows detecting whether a piece is a white piece or a black piece. A plastic piece which is white is also directly send to the conveyor. The new behavior of the crane component is presented in Figure 7.17. The automata is similar to the one previously discussed. However, transition 77 is fired if the `WP` is plastic and white. If instead the working piece is a black plastic `WP`, transition 49 is fired since the piece must be send to the conveyor.

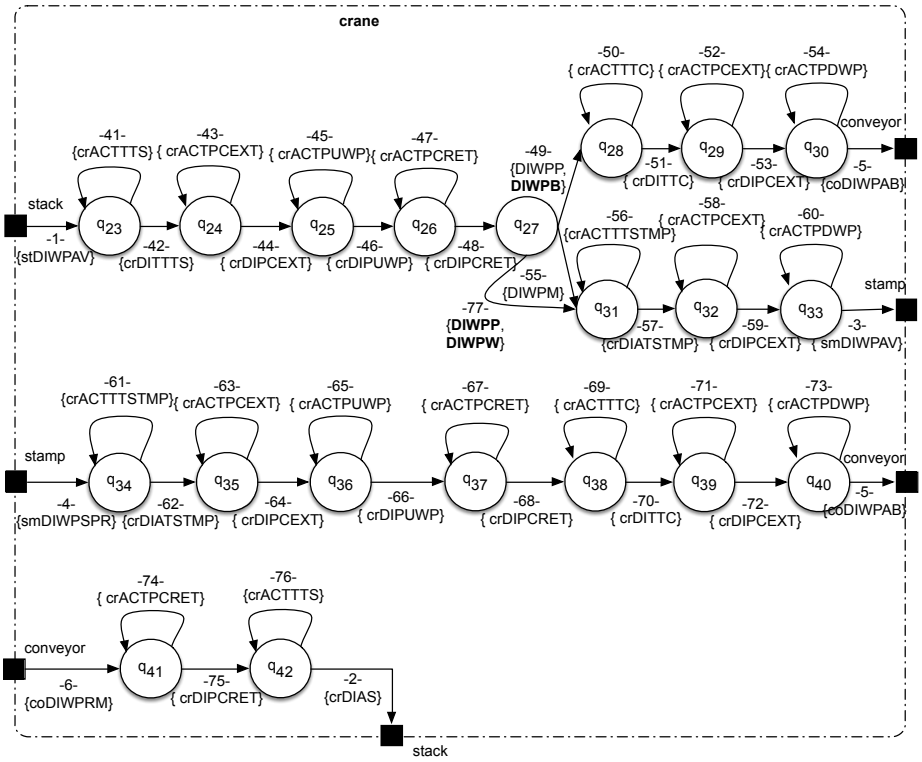


Figure 7.17: *The* `Crane` *behavior proposed in the second refinement Scenario.*

As previously, after the replacement has been designed, the developer *checks* whether the properties of interest are satisfied, possibly satisfied or not satisfied by the refined automaton. `CHIA` notifies the developer that all the properties of interest are still satisfied by the refined automaton.

CHAPTER 8

Evaluation

"Today's scientists have substituted mathematics for experiments, and they wander off through equation after equation, and eventually build a structure which has no relation to reality."

Nikola Tesla, 1856-1943

This chapter reviews the solution proposed in this work over four evaluation lines: *a*) Section 8.1 discusses the complexity of the different procedures proposed in this thesis and compares the replacement verification procedure discussed in Section 5.3 and the verification of the corresponding refinement; *b*) Section 8.2 analyzes the scalability of the approach by considering a set of random models with increasing size. It evaluates the difference in terms of time and space between checking the replacement against the previously generated constraint (the corresponding sub-property) and the effort necessary to check the refined model (the original model in which the new component is plugged into the box); *c*) Section 8.3 discusses the advantages and the weaknesses of the approach encountered during the analysis of the case studies described in Chapter 7; *d*) Section 8.4 evaluates the contributions of this thesis against the state of the art by breaking the assessment down into the following items: the modeling formalism to specify incomplete designs, the model checking algorithm, the constraint computation procedure and the replacement verification.

8.1 Complexity analysis

This section discusses the complexity of the incomplete model checking, constraint computation and replacement verification algorithms proposed in Chapter 5. It summarizes the results discussed in Section 5 for each of these operations.

8.1.1 Incomplete Model checking

The incomplete model checking algorithm presented in Section 5.1 allows the verification of an incomplete model $\mathcal{M}$ expressed as an IBA against a property expresses as a BA. As specified in Section 3.2 the BA corresponding to the property can be designed directly or obtained from an LTL formula ϕ. In the second case, given an LTL formula ϕ, the procedure to convert the formula $\neg\phi$, into the corresponding automaton $\overline{\mathcal{A}_\phi}$ has a temporal complexity $\mathcal{O}(2^{(|\neg\phi|)})$, where $|\neg\phi|$ is the size of the formula $\neg\phi$. As the property is specified with an automaton $\overline{\mathcal{A}_\phi}$, the model checking procedure presented in Section 5.1.2 in the worst case computes two intersection automata, i.e., an *under* approximation and an *over* approximation. The under approximation is computed by considering the automaton $\mathcal{M}_c$ which contains the set of behaviors the system is going to exhibit at run time. The automaton $\mathcal{M}_c$ can be obtained by removing from the automaton $\mathcal{M}$ its black box states and their incoming and outgoing transitions with a temporal complexity $\mathcal{O}(|Q_{\mathcal{M}_c}| + |\Delta_{\mathcal{M}_c}|)$. Given a sub-property $\overline{\mathcal{A}_\phi}$ with $|Q_{\overline{\mathcal{A}_\phi}}|$ states and $|\Delta_{\overline{\mathcal{A}_\phi}}|$ transitions, and a model $\mathcal{M}_c$ with $|Q_{\mathcal{M}_c}|$ states and $|\Delta_{\mathcal{M}_c}|$ the intersection automaton $\mathcal{I}_c$ has in the worst case $|Q_{\mathcal{M}_c}| \cdot |Q_{\overline{\mathcal{A}_\phi}}|$ states and $|\Delta_{\mathcal{M}_c}| \cdot |\Delta_{\overline{\mathcal{A}_\phi}}|$ transitions. The emptiness checking procedure used to verify the presence of violating runs in the automaton $\mathcal{I}_c$ has a temporal complexity $\mathcal{O}(|Q_{\mathcal{I}_c}| + |\Delta_{\mathcal{I}_c}|)$, where $|Q_{\mathcal{I}_c}|$ and $|\Delta_{\mathcal{I}_c}|$ is the number of the states and transitions of the intersection automaton. Thus, the presence of violating behaviors can be performed in time $\mathcal{O}(|Q_{\mathcal{M}_c}| \cdot |Q_{\overline{\mathcal{A}_\phi}}| + |\Delta_{\mathcal{M}_c}| \cdot |\Delta_{\overline{\mathcal{A}_\phi}}|)$. The over approximation is used to check the presence of possibly violating behaviors and is computed by considering the automaton $\overline{\mathcal{A}_\phi}$ of the sub-property against the model $\mathcal{M}$ as specified in Section 5.1.1. As proved by Theorem 5.1.1, given a sub-property $\overline{\mathcal{A}_\phi}$ with $|Q_{\overline{\mathcal{A}_\phi}}|$ states and $|\Delta_{\overline{\mathcal{A}_\phi}}|$ transitions, and a model $\mathcal{M}$ with $|R_\mathcal{M}|$ regular states, $|B_\mathcal{M}|$ boxes and $|\Delta_\mathcal{M}|$ transitions, the intersection automaton has in the worst case $\mathcal{O}((|R_\mathcal{M}| + |B_\mathcal{M}|) \cdot |Q_{\overline{\mathcal{A}_\phi}}|)$ states and $\mathcal{O}((|\Delta_\mathcal{M}| + |B_\mathcal{M}|) \cdot |\Delta_{\overline{\mathcal{A}_\phi}}|)$ transitions. As in the previous case, the emptiness checking procedure used to verify the presence of violating runs in the automaton $\mathcal{I}$ has a temporal complexity $\mathcal{O}(|Q_\mathcal{I}| + |\Delta_\mathcal{I}|)$. Thus, the presence of possibly violating behaviors can be performed in time $\mathcal{O}((|R_\mathcal{M}| + |B_\mathcal{M}|) \cdot |Q_{\overline{\mathcal{A}_\phi}}| + (|\Delta_\mathcal{M}| + |B_\mathcal{M}|) \cdot |\Delta_{\overline{\mathcal{A}_\phi}}|) \approx \mathcal{O}(|\mathcal{M}| \cdot |\overline{\mathcal{A}_\phi}|)$, where $|\mathcal{M}|$ and $|\overline{\mathcal{A}_\phi}|$ are the size of the automaton associated with the model and the claim, respectively. This implies that the incomplete model checking procedure has a temporal complexity $\mathcal{O}(|\mathcal{M}| \cdot |\overline{\mathcal{A}_\phi}|)$.

8.1.2 Constraint computation

The constraint computation phase aims at identifying for each box the corresponding sub-property. As described in Section 5.2 the constraint computation is made out of two phases: intersection cleaning and sub-properties generation.

The *intersection cleaning* takes as input the intersection automaton $\mathcal{I}$ between the model $\mathcal{M}$ and the automaton $\overline{\mathcal{A}_\phi}$ associated with the negation of the property of interest and produces the automaton $\mathcal{I}_\Upsilon$, i.e., a version of the intersection automaton where the states from which it is not possible to reach an accepting state which can be entered infinitely many often are removed. As demonstrated in Theorem 5.2.1 the cleaning procedure can be performed in time $\mathcal{O}(|Q_\mathcal{I}| + |\Delta_\mathcal{I}|)$.

The *sub-property generation* step can in turn be divided in three sub-steps: automata extraction, computation of the Π function (which computes the labels for the incoming

and outgoing transitions) and computation of the under and over reachabilty relation.

The automata extraction computes the automata associated with the different boxes and their incoming and outgoing transitions. As proved in Theorem 5.2.2 the automata extraction process can be performed in time $\mathcal{O}(|Q_{\mathcal{I}_\Upsilon}| + |\Delta_{\mathcal{I}_\Upsilon}|)$.

The computation of the Π function aims at associating to the incoming/outgoing transitions of the sub-property a value that specifies whether the initial/accepting state is directly reachable from the incoming/outgoing transitions through a run that only contains purely regular states, or it is necessary to traverse other mixed states i.e., the replacement of other boxes. The procedure proposed in Section 5.2.2 has a $\mathcal{O}(|Q_{\mathcal{I}_\Upsilon}| + |\Delta_{\mathcal{I}_\Upsilon}|)$ temporal complexity and must be executed for each sub-property $\overline{\mathcal{S}_b}$ associated with the box $b \in B_\mathcal{M}$ leading to a $\mathcal{O}(|B_\mathcal{M}| \cdot (|Q_{\mathcal{I}_\Upsilon}| + |\Delta_{\mathcal{I}_\Upsilon}|))$ temporal complexity. The function $\mathcal{U}$ which specifies for each box b whether there exists a run in the intersection automaton that possibly satisfies the claim and does not involve b can be computed by performing $|B_\mathcal{M}|$ emptiness checking procedures. Each time the emptiness procedure is performed on a variation of the intersection automaton where the mixed states generated by the box b are removed. Thus, the procedure has a $\mathcal{O}(|B_\mathcal{M}| \cdot (|Q_{\mathcal{I}_\Upsilon}| + |\Delta_{\mathcal{I}_\Upsilon}|))$ temporal complexity.

Let us now evaluate the complexity of computing the under and over reachability relations. Given the set of transitions $|\Delta_{\mathcal{I}_\Upsilon}|$ of the intersection automaton in the worst case every transition is an incoming or an outgoing transition of a sub-property. Propositions 5.2.1 and 5.2.2 imply that in the worst case scenario the reachability relations contain $\frac{|\Delta_{\mathcal{I}_\Upsilon}|}{2} \cdot \frac{|\Delta_{\mathcal{I}_\Upsilon}|}{2}$ transitions. For Theorem 5.2.7 the complexity of the algorihtm used to compute this relation is $\mathcal{O}(|Q^3| + \frac{|\Delta_{\mathcal{I}_\Upsilon}|}{2} \cdot \frac{|\Delta_{\mathcal{I}_\Upsilon}|}{2})$. Note that this complexity is acceptable since it refers to an operation that is performed at design time. The developer may accept an additional overhead in exchange of a lightweight replacement verification, where the developer wants immediate responses over his/her design.

Note that given a box b with $|\Delta_b^-|$ and $|\Delta_b^+|$ incoming and outgoing transitions the number of incoming and outgoing transitions of the sub-property $\overline{S_b}$ is in the worst case $|\Delta_b^-| \cdot |\Delta_{\overline{\mathcal{A}_\phi}}|$ and $|\Delta_b^+| \cdot |\Delta_{\overline{\mathcal{A}_\phi}}|$, respectively. Thus, given a box b, the reachability relation contains in the worst case $|\Delta_b^-| \cdot |\Delta_{\overline{\mathcal{A}_\phi}}| \cdot |\Delta_b^+| \cdot |\Delta_{\overline{\mathcal{A}_\phi}}|$ reachability entries.

8.1.3 Replacement checking

The replacement checking procedure considers a replacement $\mathcal{R}_b$ of the box b against the corresponding constraint $\mathcal{C}$. It generates an under and an over approximation of the intersection automaton computed as described in Definition 5.3.2. Under and over approximation decorate the intersection automaton using the reachability relation and the values associated to each incoming and outgoing transition to encode the behaviors that do not satisfy and possibly satisfy the properties of interest.

As proved in Lemma 5.3.5, the complexity of the replacement checking procedure is $\mathcal{O}(|Q_{\mathcal{R}_b}| \cdot |Q_{\overline{\mathcal{P}_b}}| + |\Delta_{\mathcal{R}_b}| \cdot |\Delta_{\overline{\mathcal{P}_b}}| + |\Delta^{inR_b}| \cdot |\Delta^{inS_b}| + |\Delta^{outR_b}| \cdot |\Delta^{outS_b}| + (|\Delta^{out\overline{\mathcal{S}_b}}| \cdot |\Delta^{in\overline{\mathcal{S}_b}}|) \cdot (|\Delta^{outR_b}| \cdot |\Delta^{inR_b}|))$ since it depends on the size of the over and under approximation of the intersection automaton.

The term $|Q_{\mathcal{R}_b}| \cdot |Q_{\overline{\mathcal{P}_b}}| + |\Delta_{\mathcal{R}_b}| \cdot |\Delta_{\overline{\mathcal{P}_b}}|$ specifies the size of the intersection between the automaton associated with the replacement and the automaton associated with the sub-property. Note that each state of this automaton is obtained by combin-

ing a state of the sub-property, which has the form $\langle b_{\mathcal{M}}, p_{\overline{\mathcal{P}_b}}, x\rangle$, where $b_{\mathcal{M}}$ is a box of $\mathcal{M}$ and $p_{\overline{\mathcal{P}_b}}$ is a state of the sub-property $\overline{\mathcal{A}_\phi}$, with a state $q_{\mathcal{R}_b}$ of the replacement, resulting in a state $\langle\langle b_{\mathcal{M}}, p_{\overline{\mathcal{P}_b}}, x\rangle, q_{\mathcal{R}_b}, y\rangle$, which corresponds to a state $\langle q_{\mathcal{N}}, p_{\overline{\mathcal{P}_b}}, y\rangle$ of the intersection obtained by considering the refined automaton $\mathcal{N}$. Note that given a state $\langle q_{\mathcal{N}}, p_{\overline{\mathcal{P}_b}}, y\rangle$ generated in the replacement checking procedure, in the worst case, three states $\langle\langle b_{\mathcal{M}}, p, x\rangle, q_{\mathcal{R}_b}, y\rangle$ can be generated in the replacement checking, where x is associated to 0, 1 and 2, respectively.

The terms $|\Delta^{inR_b}| \cdot |\Delta^{in\overline{\mathcal{S}_b}}|$ and $|\Delta^{outR_b}| \cdot |\Delta^{out\overline{\mathcal{S}_b}}|$ refer to the transitions that go from the artificially injected initial states g and y_i to the states of the intersection automaton and from the state of the intersection automaton to the artificially injected accepting states r and y_a. In the worst case, every incoming/outgoing transition of the replacement can be associated with every incoming/outgoing transition of the sub-property. Note that an incoming transition $\delta^{in\overline{\mathcal{S}}} \in \Delta^{in\overline{\mathcal{S}_b}}$ is generated starting from an incoming transition $\delta^-_{\mathcal{M}} \in \Delta^-_b$ of $\mathcal{M}$ and a transition $\delta_{\overline{\mathcal{A}_\phi}}$ of $\Delta_{\overline{\mathcal{A}_\phi}}$. This implies that in the worst case (when all the transitions of the automata are incoming/outgoing transitions of the boxes) $|\Delta^{in\overline{\mathcal{S}_b}}| = |\Delta_{\mathcal{M}}| \cdot |\Delta_{\overline{\mathcal{A}_\phi}}|$ and $|\Delta^{out\overline{\mathcal{S}_b}}| = |\Delta_{\mathcal{M}}| \cdot |\Delta_{\overline{\mathcal{A}_\phi}}|$. The total number of transitions from/to the artificially injected states are $|\Delta^{in\overline{\mathcal{S}_b}}| \cdot |\Delta^{inR_b}|$ and $|\Delta^{out\overline{\mathcal{S}_b}}| \cdot |\Delta^{outR_b}|$, i.e., $.|\Delta_{\mathcal{M}}| \cdot |\Delta_{\overline{\mathcal{A}_\phi}}| \cdot |\Delta^{inR_b}|$ and $|\Delta_{\mathcal{M}}| \cdot |\Delta_{\overline{\mathcal{A}_\phi}}| \cdot |\Delta^{outR_b}|$. These values are higher compared to the one necessary to check the replacement against the corresponding sub-property, where $|\Delta_{\overline{\mathcal{A}_\phi}}| \cdot |\Delta^{inR_b}|$ and $|\Delta_{\overline{\mathcal{A}_\phi}}| \cdot |\Delta^{outR_b}|$ transitions are generated with respect to the incoming and outgoing transitions of $\mathcal{R}_b$.

Finally, the term $(|\Delta^{out\overline{\mathcal{S}_b}}| \cdot |\Delta^{in\overline{\mathcal{S}_b}}|) \cdot (|\Delta^{outR_b}| \cdot |\Delta^{inR_b}|))$ concerns the number of transitions injected due to reachability relation which allows to abstract the state space "surrounding" the intersection between the replacement and the sub-property of interest. In the worst case, every couple of outgoing and incoming transition in $\Delta^{out\overline{\mathcal{S}_b}} \times \Delta^{in\overline{\mathcal{S}_b}}$ can be associated with every couple of incoming and outgoing transition in $|\Delta^{outR_b}| \cdot |\Delta^{inR_b}|$ leading to $(|\Delta^{out\overline{\mathcal{S}_b}}| \cdot |\Delta^{in\overline{\mathcal{S}_b}}|) \cdot (|\Delta^{outR_b}| \cdot |\Delta^{inR_b}|))$ transitions. Note that however, the size of $|\Delta^{outR_b}|$ and $|\Delta^{inR_b}|$ depends on the number of incoming and outgoing transitions of the boxes which is usually small compared to the sizes of the automata. Furthermore, these relations allow the model checking procedure to not compute huge portions of the intersection automaton.

8.2 Scalability assessment

Whenever a new replacement $\mathcal{R}_b$ for the box b is proposed, the developer can check the refined automaton $\mathcal{N}$ against the automaton $\overline{\mathcal{A}_\phi}$ associated with the original sub-property ϕ (refinement checking), or the replacement $\mathcal{R}_b$ against the corresponding sub-property $\overline{\mathcal{S}_b}$ (replacement checking). This section compares the efficiency of the refinement verification (Section 5.1) against the replacement checking (Section 5.3). In the absence of a realistic benchmark suite, we have *randomly* generated a set IBAs and replacements. By controlling the density of transitions, of the accepting states and of the boxes we have analyzed the performances of the algorithms over different types of models. The evaluation analyzes the configurations (values of the parameters) for which the verification of the replacement outperforms the verification of the refinement of the model.

8.2.1 Experimental setup

To evaluate the applicability of the approach, in absence of a realistic benchmark suite, we have *randomly* generated a set IBAs representing the models of the system which are considered against a set of predefined LTL properties.

The IBAs generation is based on the procedure presented in [127] (and also used in [126], [38]), that has been demonstrated to provide an interesting source of benchmark problems. In [127], a random Büchi automaton is generated over an alphabet made by two propositions, e.g., $\{a, b\}$. For each proposition of the alphabet a random directed graph with a single initial state and k transition is generated. The "hardness" of the problem is changed by controlling the density of the *transitions*, i.e., the ratio between the number of transitions per proposition p and the number of the states of the automaton ($r = |\Delta_p|/|Q|$, where $|\Delta_p|$ is the cardinality of the set of transitions labeled with p), and the density of *accepting* states, i.e., the ratio between the accepting states and the number of the states of the automaton ($f = |F|/|Q|$). The transition density represents the expected number of transitions that exits a state. The number of accepting states, which are selected randomly between the set of the states of the BA, is a linear function of the number of the states of the system. The generation procedure also imposes the initial state to have an outgoing transition for each proposition of the alphabet, to avoid the generation of trivial automata.

The random generation procedure used in our assessment uses the BA obtained with the procedure previously described to extract both the IBAs and the replacements of the boxes. The idea is to randomly encapsulate parts of the BAs into boxes. The encapsulated automata and the corresponding incoming and outgoing transitions are the replacements associated with the boxes, while the IBA is the automaton obtained by replacing the encapsulated parts with the corresponding boxes. The box density ($b = |B|/|Q|$) specifies the number of boxes which must be injected into the BA. The replacement density ($r = (|B| \cdot |Q_b|)/|Q|$) specifies the number of states of the automaton to be encapsulated inside the replacement of the boxes. The states are chosen between the states of the BA and randomly inserted into the replacement of one of the boxes. Given a state s which is encapsulated into a box b, each incoming and outgoing transition of s is associated with an incoming and outgoing transition of the replacement $\mathcal{R}_b$ of the box b. Furthermore, each incoming (outgoing) transition with wource (destination) s' of s is associated with an incoming (outgoing) transition of b with source (destination) s'.

Parameter	Initial Value	Increment	Final Value
N°of states	10	10	100
Transition density	1.0	1.0	4.0
Accepting density	0.2	0.1	0.5
Box density	0.1	0.1	0.5
Replacement density	0.1	0.1	0.5

Table 8.1: *Values of the parameters used in the scalability assessment.*

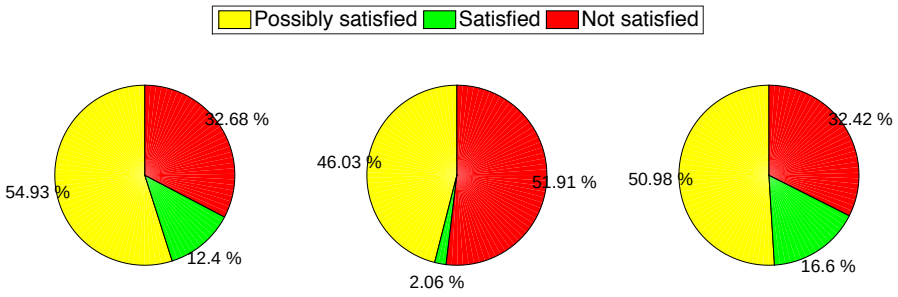
The values of the parameters used in the scalability assessment are presented in Table 8.1. The transition density changes from 1.0 to 4.0 with a unitary increment. The transition density specifies, for each proposition, the average number of transitions

that exit each state of the model. For example, a transition density of 4 means that, in average, each state of the system is left from 4 transitions per proposition. The range of the accepting density is between 0.2, which indicate that two states over ten are accepting, and 0.5, which states that half of the states are accepting. The number of states of the model changes from 10 to 100. To transform BA into IBA the values of the box density and the replacement density presented in Table 8.1 are considered. Given a BA with n states a box density of 0.5 specifies that $n/2$ boxes are added to the BA. The replacement density is the ratio between the sum of the states injected into the replacements of the boxes and the number of the states of the BA. When the replacement density is equal to 0.5, the replacements of the boxes contain half of the states of the BA.

Three claims are considered in the scalability evaluation: the LTL formula $\phi_1 = (a)U(b)$, the safety formula $\phi_2 = G(a \rightarrow (F(b)))$ and the liveness formula $\phi_3 = F((a)U(b))$. For each of these formulae, and for each configuration generated as specified in Table 8.1, 20 tests have been performed, leading to 240.000 configurations.

8.2.2 Experimental results

The tool is evalutated over the $240^{'}000$ configurations generated as specified in Section 8.2.1, over the three LTL properties ϕ_1, ϕ_2 and ϕ_3. When a randomly generated model is considered with respect to one of these properties, the model checker may return three possible values, satisfied (T), possibly satisfied ($\perp$) and not satisfied (F). Figure 8.1 specifies for each LTL formula the percentage of the cases in which the property is satisfied, possibly satisfied or not satisfied. Note that, these percentages depend both on the type of the property which has been considered and on the algorithm which is used to randomly generate the models to be considered. Notice that the percentage of the cases in which the formula $(a)U(b)$ is satisfied is less than the percentage of the percentage in the case in which $F((a)U(b))$ is considered. Indeed, if a model satisfies the property $(a)U(b)$ it also satisfies $F((a)U(b))$.



(b) *Satisfaction of the claim ϕ_1.* **(c)** *Satisfaction of the claim ϕ_2.* **(d)** *Satisfaction of the claim ϕ_3.*

Figure 8.1: *The model checking results obtained by considering the claims ϕ_1, ϕ_2 and ϕ_3.*

The relation between the values returned by the model checker and some of the parameters specified in Table 8.1 is described in the following.

First, we have considered the relation between the percentage of the models that satisfy, do not satisfy and possibly satisfy the property of interest and the number of the states of the automaton. The relation has a similar behavior with respect to all the properties of interest. Figure 8.2 relates the number of the states and the satisfaction values

when the property ϕ_1 is considered. The percentage of models that possibly satisfy the property increases as the number of the states of the model increases. Indeed, the number of boxes injected into the model increases linearly with the number of states of the generated automaton. This implies that, the more states are in the models, the higher is the chance for the property to be possibly satisfied. Since percentages can assume a maximum value of 100%, we checked the presence of a monotonic relation between the number of the states and the possibly accepting percentage using the Spearman's rank correlation coefficient[1].

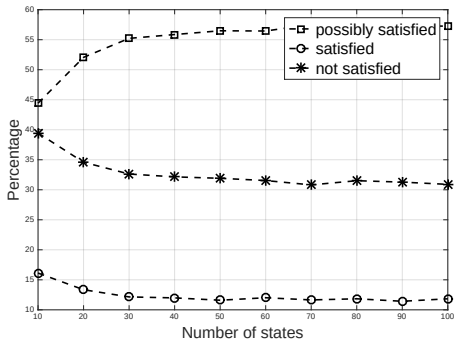


Figure 8.2: *Satisfaction of ϕ_1 with respect to the number of the states.*

The Spearman's rank correlation coefficient assesses how well the relationship between two variables can be described using a monotonic function. A perfect Spearman correlation of $+1$ or -1 occurs when each of the variables is a perfect monotone function of the other. The relation between the number of the states and the possibly accepting percentage has a Spearman's rank correlation coefficient close to 1.

The percentage of the models that satisfy, do not satisfy and possibly satisfy the properties of interest with respect to the states injected inside the replacement of the boxes is presented in Figure 8.3, where the property ϕ_1 is considered. The percentage of models that possibly accept the property of interest increases as the density of the replacements of the boxes increases and has a monotonic shape has confirmed by the Spearman correlation coefficient. Indeed, the higher is the number of the states encapsulated into the replacement of the boxes, the lower is the number of the regular states

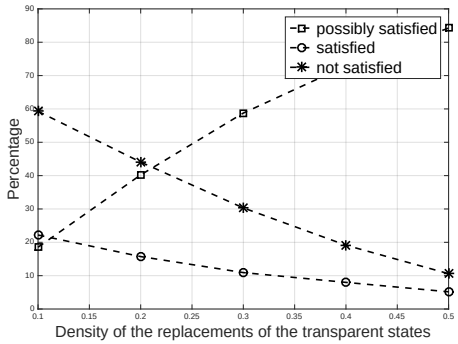


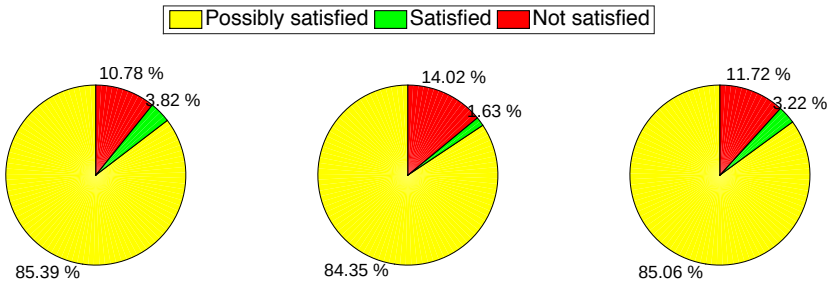
Figure 8.3: *Satisfaction of ϕ_1 with respect to the number of the box states.*

[1]The presence of an upper-bound on the percentage values makes the Pearson correlation coefficient [99] not suitable for the analysis of the relation between the number of the states and the possibly accepting percentage.

that “remains" in the IBA. For example, when half of the states of the automaton are encapsulated into the boxes (i.e., the replacement density is equal to 0.5), the property is possibly satisfied in the 84.20% of the cases.

Over the $240\,000$ experiments, in $24\,994$, $121\,624$, $93\,382$ cases the properties ϕ_1, ϕ_2 and ϕ_3 were satisfied, possibly satisfied and not satisfied, respectively. Whenever the property is *possibly satisfied* the replacement for one of the boxes of the model is considered. The replacement is used to compare the performance of its verification against the corresponding constraints and the verification of the claim against the refinement of the system. In the first case, the constraint computation algorithm described in Section 5.2 is used to compute from the model of the system and the corresponding claim a set of constraints the replacement of the boxes must satisfy. The performances of the replacement checking activity concerns the time and space necessary to verify whether the replacement satisfies the corresponding constraint. In the second case, the replacement is first injected into the original model, and the obtained refinement is checked against the claims of interest. The comparison between the two approaches is performed by considering two different measures: *time* and *space*. The time concerns the duration in seconds of the model checking activity. The space is the size of the automata generated in the verification.

In all the $121\,624$ cases in which the property was *possibly satisfied*, the experiments confirm the correspondence between the results of the verification of the replacement and of the refinement, which prompts a correct implementation of the checkers. The percentage of the cases in which the property was satisfied, not satisfied and possibly satisfied for each LTL property in the replacement verification is described in Figure 8.4. As evidenced in Figure 8.4, in most of the cases in which a replacement is injected into a box, the property remains possibly satisfied. This because it is highly probable that the automaton to be refined contains possibly accepting runs which do not involve the refined box b, or the possibly accepting runs involving b contain also other boxes.



(b) *Satisfaction of the claim* ϕ_1. **(c)** *Satisfaction of the claim* ϕ_2. **(d)** *Satisfaction of the claim* ϕ_3.

Figure 8.4: *The model checking results obtained in the replacement verification considering the claims* ϕ_1, ϕ_2 *and* ϕ_3.

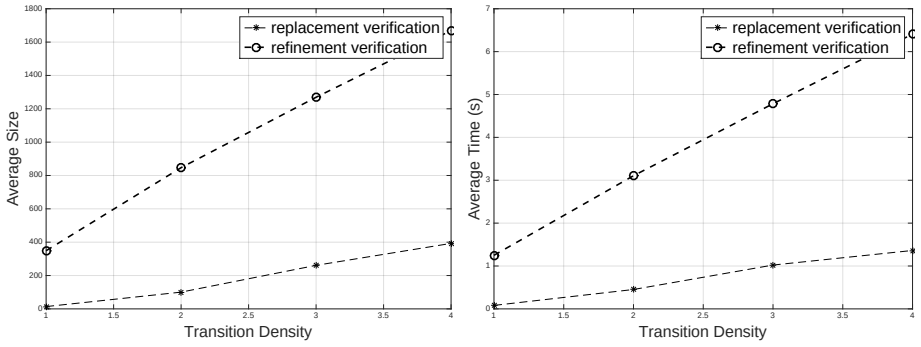
In the $44\,062$, $36\,859$ and $40\,703$ cases in which the property ϕ_1, ϕ_2 and ϕ_3 were possibly satisfied, respectively, the replacement verification outperforms the refinement verification in the 95.45%, 93.56% and 96.15% of the cases with respect to the verification time and in the 92.52%, 93.26% and 92.99% of the cases with respect to the verification space. The average and the standard deviation of the verification time and the size of the automata generated in the refinement and in the replacement checking

is represented in Table 8.2. The table evidences that the replacement verification generates automata which in average are smaller than the one generated by the refinement verification. However, there is a higher variance in the size of the automata generated (i.e., in the worst case the reachability relation has a polynomial impact on the size of the automata). Furthermore, the time required by the replacement verification procedure (seconds) is, in average, less than the one required by the refinement verification. However, as for the size, the higher variance specifies that there are cases in which there is a high overhead introduced by the replacement verification.

		$\phi_1 = (a)U(b)$		$\phi_2 = G(a \rightarrow (F(b)))$		$\phi_3 = F((a)U(b))$	
		M	SD	M	SD	M	SD
Ref	Size	991.94	854.04	949.57	752.49	348.99	295.85
	Time	3.72	3.38	3.44	2.94	1.70	1.45
Rep	Size	179.16	3655.02	174.61	1247.55	56.37	535.03
	Time	0.69	12.85	0.67	4.74	0.21	1.82

Table 8.2: *The average (M_a) and the variance (SD) on the size of the automata generated and the time required by the refinement (Ref) and the replacement (Rep) verification procedures.*

The relation between the average size of the automata generated by the replacement and the refinement verification procedure with respect to the density of the transitions of the automata is presented in Figure 8.5, where the property ϕ_1 is considered.



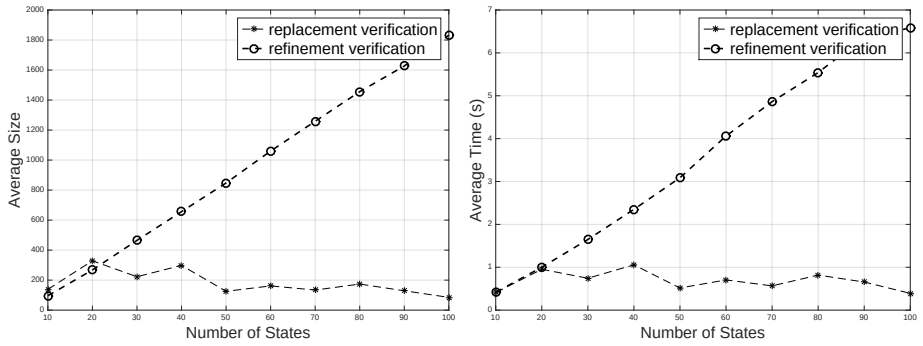
(a) *Average size.* **(b)** *Average time.*

Figure 8.5: *Average size of the automata generated and time required by the refinement and the replacement verification procedures with respect to the transition density when the property ϕ_1 is considered.*

As demonstrated in Section 8.1.3, the complexity of the replacement verification procedure has a polynomial overhead in the number of the incoming/outgoing transitions of the replacement. However, the trends of Figure 8.5a specify that in most of the cases, the reachability between the incoming and outgoing transitions is not computed. The refinement verification procedure (identified in Figure with a dotted line marked through circles) increases linearly with respect to the density of the transitions of the automaton. Instead, the increment in the transition density has a polynomial impact on the replacement verification (identified in Figure with a dotted line marked through stars). However, due to the presence of multiple boxes, in many cases, the property is possibly satisfied since the incoming transitions and outgoing transitions of the replacement are not “directly" reachable from the initial and the accepting state of the

IBA. This means that the replacements of other boxes must be traversed before reaching the considered box. In all these cases the reachability relation is not used in the replacement verification.

The relation between the size of the automata and the time required by the replacement and refinement verification procedures and the number of the states the model of the system, when the property ϕ_1 is considered, is presented in Figure 8.6. As evidenced in Figure 8.6a, while the verification of the size of the automata generated in the refinement verification (identified in Figure with a dotted line marked through circles) increases linearly with the size of the automata to be considered, in the replacement case (identified in Figure with a dotted line marked through stars) the size of the automata decrease. The same behavior is shown by the verification time. Indeed, the number of boxes increases linearly with the automata size and, as the size of the automata increases, the probability of having accepting runs which involve other boxes increases. When the incoming transitions of the box which is considered are not “directly" reachable from one of the initial states of the automaton, and from the outgoing transitions it is not possible to “directly" reach an accepting state, the property is possibly satisfied and, in all these cases, it is not necessary to compute any intersection leading to high performance improvements.

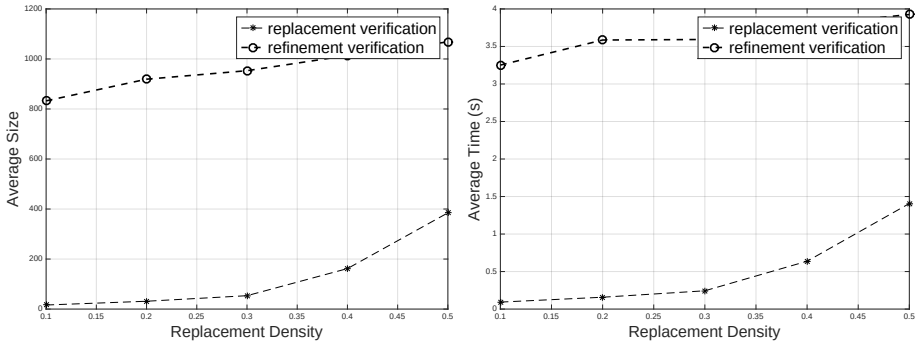


(a) *Average size.* **(b)** *Average time.*

Figure 8.6: *Average size of the automata generated and time required by the refinement and the replacement verification procedures with respect to the number of the states of the automata when the property ϕ_1 is considered.*

The relation between the size of the automata generated and the time required by the replacement and refinement verification procedures and the replacement density is presented in Figure 8.7. The refinement verification is barely affected by the replacement density: the number of the states encapsulated into a box influence the refinement verification time since the corresponding refined automaton is generated. Conversely, the number of the states of the replacement has a strong influence on the replacement verification. Indeed, the higher is the number of the states encapsulated into replacements the higher is the probability of connecting incoming and outgoing transitions, making the use of the reachability relation necessary.

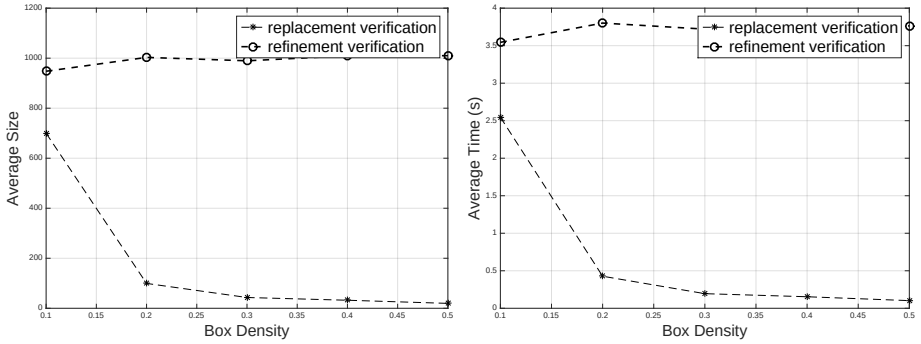
Finally, Figure 8.8 presents the average size of the automata generated and the time required by the refinement and the replacement verification procedures with respect to the box density when the property ϕ_1 is considered. Again, the refinement verification is barely affected by the box density: the number of the boxes injected influence the



(a) *Average size.* **(b)** *Average time.*

Figure 8.7: *Average size of the automata generated and time required by the refinement and the replacement verification procedures with respect to the replacement density, when the property ϕ_1 is considered.*

refinement verification time since they are added to the states of the automaton. Conversely, the sizes of the automata generated in the replacement verification decreases, until asintotically, the size of the automata generated is three. The reason is the following, by increasing the number of boxes, there is a high chance that the box b which is refined can be reached from the initial state by runs that traverse other boxes and, similarly, can reach the accepting states only by traversing other boxes. Thus, all the incoming and outgoing transitions of the sub-property associated to the box b are marked as "Y". Furthermore, there is a high chance that there exist a (set of) possibly accepting run(s) which do not involve the refinement of the box currently analyzed. For these reasons, the lower approximation automaton has a size 3, i.e., the lower approximation only contains the artificially injected initial (g) the accepting (r) state and the stuttering transition over the accepting state.



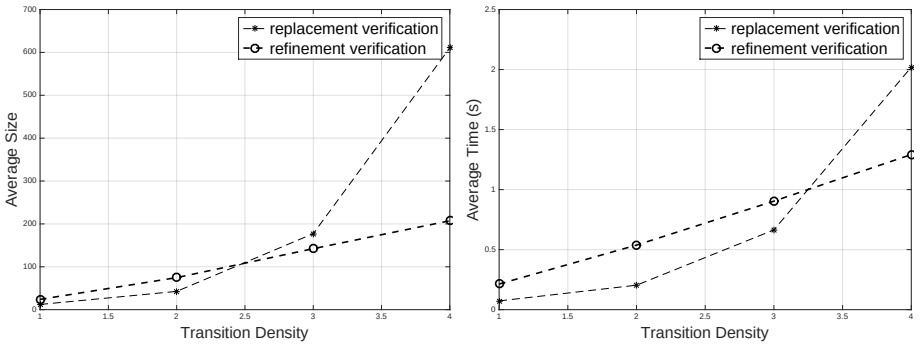
(a) *Average size.* **(b)** *Average time.*

Figure 8.8: *Average size of the automata generated and time required by the refinement and the replacement verification procedures with respect to the box density when the property ϕ_1 is considered.*

To analyze the effectiveness of the approach in the worst case, we consider only the experiments in which the intersection between the replacement and the refinement is computed and the property results not satisfied. In the $4\,704$, $5\,119$ and $4\,744$ cases in which the refinement does not satisfy the property of interest the replacement verifica-

tion was faster in the 64.58%, 58.00% and 66.17% of the cases. The diagrams presented in the following refers to the property ϕ_1. Similar shapes have been obtained when the properties ϕ_2 and ϕ_3 have been considered.

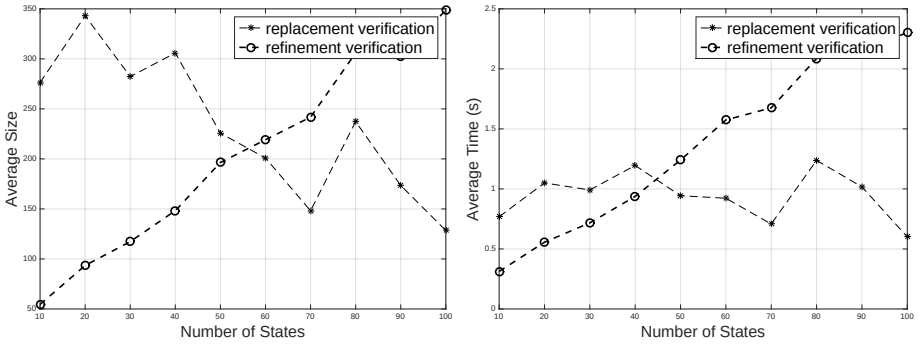
Figure 8.9 describes the average size of the automata generated with respect to the transition density. The figure evidences the quadratic impact of the reachability relation on the size of the automata generated in the replacement verification. Note that, when the transition density is low, the replacement outperforms the refinement verification, while when the density of the transitions increases the refinement verification is more effective.



(a) *Average size.* **(b)** *Average time.*

Figure 8.9: *Average size of the automata generated and time required by the refinement and the replacement verification procedures with respect to the transition density, in the cases in which the refinement does not satisfy the property of interest.*

Figure 8.10 specifies the relation between the number of the states of the model, the average size and time required by the replacement and refinement checking. Note that, the number of the states may not correspond to the states included in the final model since some of them may be encapsulated into boxes as specified in Section 8.2.1. Figures 8.10a and 8.9b specify that the number of the states barely influence the effort required in the replacement verification. Indeed, the replacement verification depends

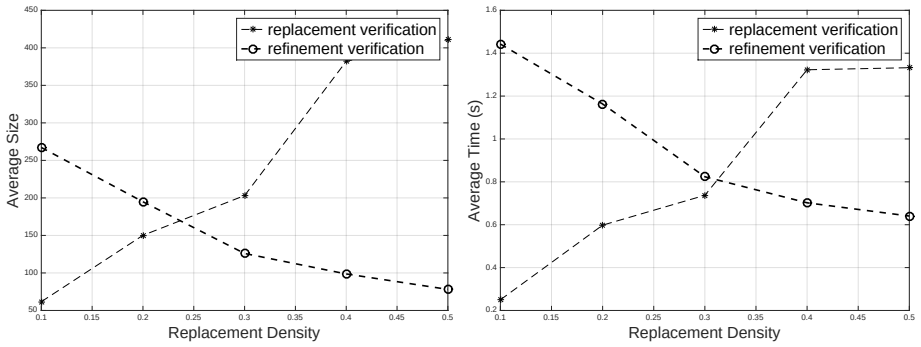


(a) *Average size.* **(b)** *Average time.*

Figure 8.10: *Average size of the automata generated by the refinement and the replacement verification procedures with respect to the number of the states of the automata when the refinement does not satisfy the property of interest.*

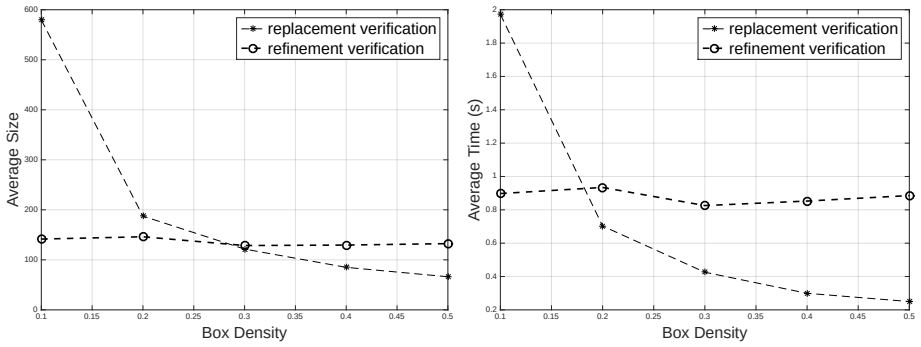
on the size of the replacement, which depends on how the states of the model are randomly encapsulated into the boxes. For this reason, and since the reachability relation is used in the generation of the lower and upper intersection automata to be checked, it is possible that for really small size models the replacement verification procedure generate big models. Conversely, the effort required in the refinement verification grows linearly with the size of the model to be checked.

Figure 8.11 describes the average size of the automata generated in the replacement and in the refinement verification and the corresponding verification time with respect to the replacement density. The effort required to check the refinement automaton decreases when the replacement density increases. The reason is the following. Consider the case in which the replacement density is 0.5. Given a set of boxes (whose number depends on the box density), half of the states of the original model are randomly encapsulated into the boxes. However, when the refinement is generated only the replacement associated with one box (the one which is considered) is injected into the model. Conversely, the replacement density has a direct impact on the replacement verification. Intuitively, the higher is the size of the replacement, the higher is the effort necessary in the replacement verification.



(a) *Average size.* **(b)** *Average time.*

Figure 8.11: *Average size of the automata generated by the refinement and the replacement verification procedures with respect to the replacement density when the refinement does not satisfy ϕ_1.*



(a) *Average size.* **(b)** *Average time.*

Figure 8.12: *Average size of the automata generated by the refinement and the replacement verification procedures with respect to the box density when the refinement does not satisfy ϕ_1.*

Finally, Figure 8.12 shows the average size of the automata generated by the refinement and the replacement verification procedures and the time required by the verification procedure with respect to the box density, in the cases in which the refinement does not satisfy the property ϕ_1 of interest. The effort required in the refinement verification is barely influenced by the box density. Conversely, when the box density increases, the average size of the automata encapsulated into each box decreases. This benefits the replacement verification and results in a lower size of the automata generated.

8.3 Case studies

The approach proposed in this thesis has been used over the two case studies described in Chapter 7. This section evaluates the approach with respect to three different aspects:

- *modeling formalism*: we discuss the advantages and limitations of the proposed formalism evidenced on its use over the two case studies described in Chapter 7;
- *constraint computation*: we describe how the constraint computed using the procedure described in Section 5.2 supports the development;
- *replacement verification*: we evaluate the performances of the replacement verification procedure presented in Section 5.3 over two realistic examples.

The IBA *modeling formalism* natively support hierarchical decomposition, encapsulation and iterative refinement. The case studies confirm the advantages of using this formalism in such a development setting, which easily allows the modeling and refinement of the features of interest. On the other hand, the case studies also reveals two main drawbacks (and possible extensions) of the formalism: *a*) sometimes the developer may want to specify an assumption, i.e., an invariant over the value of the propositions inside a specific black box state. For example, he/she may want to specify that the crane is not able to modify the value of the propositions of the stamp, i.e., $smACTSLRET$ is never true inside the crane. This is an assumption specifying that the stamp actuator is not activated by the crane, and can be performed in the cases in which the developer in charge of designing the crane has no possibility to modify the satisfaction of that proposition. The specification of invariants enriches the formalism by supporting a higher separation of concerns; *b*) IBAs only support sequential systems, i.e., they do not natively support parallel execution. The specification of concurrent systems such as the mutual exclusion system described in Section 7.1 can be performed only by generating all the possible interleaving between the actions that can be performed by the processes and encapsulating the unknown behaviors inside boxes.

The *constraint computation* algorithm introduced in Chapter 5 has been proposed to support the designed in the development activity. The two case studies presented in Chapter 7 evidence the possible advantages that descend by the use of this technique. Between the others we highlight: *a*) they are an instrument that allows a more conscious development approach. The developer before designing a component can compute and analyze the sub-properties he/she must satisfy, giving him/her an overview on the behaviors the component must not exhibit. For example, in the development of the replacement of the box b_{10} the developer knows that he/she cannot design a component where there is an infinite internal run where cr_0 and cr_1 are not satisfied; *b*) it allows

contract negotiation. Whenever the model of the system contains multiple components still to be developed, i.e., different boxes, the design of the components can be delegated to third parties. In these cases, it is essential to define the features the different components are going to exhibit. For example, in the second scenario of the mutual exclusion problem, it is important to establish whether b_{10} or b_{11} is going to allow the process P_1 to enter its critical section. On the other hand, at the current stage, the analysis of the sub-property is not a simple task. The sub-properties encode all the possible ways in which the replacement can violate the claim of interest. Interpreting such behaviors is not easy especially because the sub-properties encode through the reachability relation how the structure of the original model influence the replacement design. Furthermore, *a*) at the current development stage a graphical representation of the sub-properties is not provided by `CHIA`. The developer must use the textual description whose interpretation is cumbersome and time consuming; *b*) several reachability entries are duplicated since they are needed in the replacement verification. A more compact representation can be obtained by removing the duplicated entries.

The *replacement verification* has been used to check the replacement $\mathcal{R}_b$, of the box b, against the previously generated constraint. We have compared the performances of the verification of the refined model $\mathcal{N}$, obtained by plugging the replacement $\mathcal{R}_b$ of the refined box b into the original model $\mathcal{M}$, with respect to the verification of $\mathcal{R}_b$ against its constraint $\mathcal{C}$. This type of the evaluation provides a feedback on the performances of the procedure on two realistic case studies. As done for the scalability assessment, we compare the two approaches with respect to the size of the automata generated and the time required by the verification procedure.

The results associated with the `MUTEX` case study are presented in Table 8.3. As evidenced, the replacement verification outperforms the verification of the refinement. In the *first refinement scenario*, the sizes of the automata generated and the time of the replacement verification are comparable to the refinement verification case. In all the cases, both the lower and the upper intersection automaton are considered to check the presence of violating and possibly violating runs, respectively. When the replacement

		Req 7.1.1		Req 7.1.2		Req 7.1.3		Req 7.1.4	
		REF	REP	REF	REP	REF	REP	REF	REP
Scenario 1	Time	73	10	38	26	38	33	6	4
	Size	66	44	378	284	670	574	162	116
Scenario 2	Time	11	1	6	1	41	6	7	1
	Size	120	7	260	17	527	32	130	12

Table 8.3: *The time (ms) and the size ($|Q| + |\Delta|$) of the automata generated by the refinement (REF) and the replacement (REP) verification procedures in the two MUTEX refinement scenarios.*

is considered, the lower reachability relation contains all the possible ways in which it is possible to reach an incoming transition of the box from one of its outgoing transitions. Since b_{10} is the only box in the model of the `MUTEX`, the reachability relation abstracts all the runs of the intersection automaton which do not include states generated from the combination of b_{10} and a state of the claim. Furthermore, in this case, the upper reachability relation corresponds to the lower reachability relation since b_{10} is the only box in the system. The combination of the reachability relations with the incoming and outgoing transitions of the replacement allows the generation of smaller automata with

respect to the verification of the replacement from scratch. In the *second refinement scenario*, the situation is slightly different. When the sub-property of the state b_{11} is computed, the lower reachability relation is empty, since to reach an incoming transition of b_{11} it is necessary to travel on a run that crosses b_{10}. Furthermore, there exists a run which does not involve b_{11} which is possibly violating, i.e., a run in which the property is not satisfied inside b_{10} and does not cross b_{11}. This justifies the considerable speed up obtained by using the replacement verification procedure. In the replacement verification, the under approximation automaton does not contain any entry related with the reachability relation and the over approximation is not computed since it is already exists a run which does not involve b_{11} where the property is possibly satisfied.

The results associated with the `PPU` case study are presented in Table 8.4. The requirements 7.2.1 and 7.2.3 are not considered in the evaluation since they are already satisfied by the original IBA. In both the refinement scenarios all the states of the model $\mathcal{M}$ to be refined are regular, with the exception of the crane which is the only black box state. As previously mentioned when only one box is present in $\mathcal{M}$ the lower and the upper reachability relation coincide and contain all the possible ways in which it is possible to reach in the intersection automaton an incoming transition of the box (the crane) from an outgoing one. The reachability relation is used to create the under and

		Req 7.2.1		Req 7.2.2		Req 7.2.3		Req 7.2.4	
		REF	REP	REF	REP	REF	REP	REF	REP
Scenario 1	Time	-	-	20	7	-	-	6	4
	Size	-	-	502	174	-	-	456	148
Scenario 2	Time	-	-	20	4	-	-	6	1
	Size	-	-	502	72	-	-	456	72

Table 8.4: *The time (ms) and the size ($|Q| + |\Delta|$) of the automata generated by the refinement (REF) and the replacement (REP) verification procedures in the two PPU refinement scenarios.*

the over approximation automata as specified in Definitions 5.3.3 and 5.3.4. In both the refinement scenarios, the dimensions of the automata generated, and the verification time, is lower than the one required to verify the Requirements 7.2.2 and 7.2.3 from scratch.

8.4 Comparison with other approaches

This section compares our approach with related work. The comparison includes the expressiveness of the modeling formalism, the algorithm to check incompleteness, the procedure to synthesize the constraint and the replacement checking procedure.

8.4.1 Modeling formalisms

Several modeling formalisms have been proposed in literature to describe incomplete models. A detailed description of these formalisms has been presented in Section 2.1. This section compares IBAs with existing modeling formalisms over two different criteria: *a*) the type of the system which is possible to describe; *b*) the type of incompleteness that can be expressed.

The first characteristic we have considered is whether the modeling formalism is

suitable for modeling *sequential* or *concurrent* systems[2]. A sequential system is usually described using its input-output relation. The system changes its state with respect to the input it receives from the environment and produces outputs. Conversely, in a concurrent system, different processes (computations) are executed simultaneously, and potentially interact with each other. Usually, each of the processes is modeled through a type of FSM and the parallel execution of FSMs is defined. Different modeling formalisms, such as MTS, PKS, $\mathcal{X}$KSs and LTS$^{\uparrow}$ supports the specification of incompleteness when concurrent systems are considered. Other modeling formalisms, such as PKS and HSM are more suitable for modeling sequential processes. As specified in Chapter 4, at the current development step IBAs can only describe sequential models of computation. Indeed, the main goal of IBAs is to support the designer when a top-down development process is adopted and they are not designed for the development of concurrent systems.

		MTS	PKS	HSM	$\mathcal{X}$KSs	LTS$^{\uparrow}$	IBA
Type of system	Sequential	x	x	x	x		x
	Concurrent	x			x	x	
	Hierarchical	x*	x*	x*			x
	Reactive					x	
Type of incompleteness	Component	x*	x*	x*		x*	x
	Transition	x					
	Proposition		x		x		

Table 8.5: *Comparison between the modeling formalisms proposed to manage incomplete models.*

The second aspect which has been considered in our classification is whether the formalism is designed to support *hierarchical* development. Hierarchical development concerns the decomposition and partitioning of the system into smaller modules. At each refinement round different features of the model are analyzed, with a different level of abstraction. HSMs are a modeling formalism which has been proposed for hierarchical development. However, HSMs do natively include incompleteness, i.e., a HSM contains the specifications of all the sub-modules and how they are hierarchically related with each other. Conversely, MTSs and PKSs which natively support incompleteness have not been expressly designed for being used in a hierarchical development context, even if they can be used in this setting (for this reason the corresponding entry in Table 8.5 is also marked with the symbol *). Finally, some works consider incompleteness in the context of *reactive* systems. In a reactive system the different components interact with the environment where the system is running. In LTS$^{\uparrow}$, the incompleteness concerns the absence of knowledge about the environment in which the system is executed.

Regarding the type of incompleteness that can be modeled we identify three different types of incompleteness. First, the value of a *proposition* may not be specified. In this case, given a state or a transition labeled with a set of propositions, the developer may be uncertain about the values assumed by some of them. Incompleteness over the values of the propositions has been considered for example in $\mathcal{X}$KSs and PKS. Other modeling formalisms support the specification of incompleteness over *transitions*. In this case the developer may be uncertain on whether a transition will be present in the

[2]For further details on the differences between sequential and concurrent systems see, for example [100].

final model of the system. MTSs are one of the most used modeling formalism in this context. Finally, the last type of incompleteness we considered is over *components*. This kind of incompleteness is associated with the classical concepts of hierarchical refinement, where the behavior of the system in components (which are associated with states) is iteratively refined. As previously mentioned, HSMs are the most suitable formalism for hierarchical development. However, the formal definition of HSMs requires all the components and their hierarchical relations to be specified. MTSs [77] can be used in the specification of unknown components even if they are not intrinsically designed for this purpose. For example, a box of an IBA can be converted into an incompleteness over transitions (MTS) as specified in Figure 8.13, where Σ is the universal alphabet and the transitions 1, 2 and 3 are possible transitions. Note that, with respect to MTSs, IBAs also contain accepting states, which is a necessary requirement for the verification of fairness properties [33]. Also PKS can be used to specify unknown components. PKSs are Kripke structures in which the propositions can take a third truth value $\perp$. This offers a different level of granularity with respect to IBA, where boxes are refined into other state machines. In PKSs the refinement process concerns the definition of the truth values of the atomic proposition in the different states. Note that, a state where all the atomic propositions are uncertain can also be decomposed into different states since the refinement relation is based on the notion of a completeness pre-order. However, IBAs have been expressly proposed as a formalism that supports a hierarchical development strategy in which boxes are iteratively refined into other state machines. Finally, in LTS$^{\uparrow}$ the model of the system contains a set of transitions that can be triggered by an environment whose behavior is not defined, i.e., the incompleteness refers to a component (the environment) which is executed in parallel to the system. The set of actions Act, which label the transitions, is partitioned in two sets: one set contains the set of actions that can be triggered (controlled) by the environment, the other contains the actions that cannot be controlled by the environment and represent internal actions of the model. In this sense LTS$^{\uparrow}$ are similar to MTS, where the presence of some transitions is ensured while other transitions can be present or not into the final model depending on the behavior of the environment.

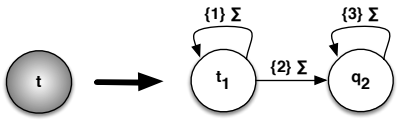


Figure 8.13: *Mapping between a box of an IBA and the corresponding MTS.*

In conclusion IBAs try to overcome the limitations of the previously mentioned modeling formalisms by proposing a framework which natively supports the developer when a top-down development style is employed.

8.4.2 Model checking

The model checking procedure for IBAs presented in this work is similar to other approaches proposed in literature. The technique presented in Section 5.1 is compared with existing frameworks which are able to consider properties expressed as LTL for-

mulae or automata[3]. In particular, we consider works that check incomplete models when a *three value inductive* semantic of LTL is considered. In this semantic, when a property is possibly satisfied on a model $\mathcal{M}$, there is no guarantee that there exists a refinement $\mathcal{N}$ of $\mathcal{M}$ where the property is satisfied and not satisfied. Differently from the inductive semantic, in the thorough semantic, a property is evaluated to maybe if there exist two refinements $\mathcal{N}$ and $\mathcal{N}'$ such that one satisfies and the other does not satisfy the property of interest[4].

	LTL	Aut
MTS	$2^{\|\phi\|+1} \cdot \|\mathcal{M}\|$	$2 \cdot \|\mathcal{A}_\phi\| \cdot \|\mathcal{M}\|$
PKS	$2^{\|\phi\|+1} \cdot \|\mathcal{M}\|$	$2 \cdot \|\mathcal{A}_\phi\| \cdot \|\mathcal{M}\|$
HSM	$-$	$-$
$\mathcal{X}$KSs	$2^{\|\phi\|+1} \cdot \|\mathcal{M}\|$	$2 \cdot \|\mathcal{A}_\phi\| \cdot \|\mathcal{M}\|$
IBA	$2^{\|\phi\|+1} \cdot \|M\|$	$2 \cdot \|\mathcal{A}_\phi\| \cdot \|\mathcal{M}\|$
LTS$^\uparrow$	$-$	$2 \cdot \|\mathcal{A}_\phi\| \cdot \|\mathcal{M}\|$

Table 8.6: *Comparison between the complexities of the model checking algorithms that support incompleteness.*

Table 8.6 shows the asymptotic behaviors of the algorithm proposed for checking automata and LTL properties over the modeling formalisms which support incompleteness described in Section 8.4.1. The asymptotic behaviors of these algorithms correspond since most of them rely on the same idea, i.e., to perform two classical model checking procedures which consider an over and an under approximation of the model of the system, respectively. The under approximation contains the behaviors the developer is sure will be exhibited by the system. If the property is violated in the under approximation it will also be violated by the final system. The over approximation contains the behaviors the final system may exhibit. If the property is violated in the over approximation it is possibly violated. Otherwise the property is satisfied.

In the case of *Modal Transition Systems* (MTSs) [129] the under approximation is obtained by removing all the maybe transitions, while in the over approximation all the maybe transitions are converted into required one. In the case of *Partial Kripke Structures* (PKSs) [11] and *$\mathcal{X}$Kripke Structures* ($\mathcal{X}$KS) [22] the under and over approximations are obtained by associating “true" and “false" values to the atomic propositions, respectively. The same idea has been used in [56] where Labeled Transition System with an additional *interface operator* ($LTS^\uparrow$) are considered. In this context, the model checking procedure can be used to check only the satisfaction of safety properties. Note that, in [56], both the model and the property are specified using automata over finite words. For further information, the interested reader may refer to Section 2.3. Finally, HSMs are marked with the symbol $-$ since, up to our knowledge, an algorithm that supports the verification of a HSM which is incomplete when a top down development approach is used is still missing. However, algorithms such as the one presented in [8] can be modified to work in this setting.

The model checking procedure for IBAs proposed in this work (Section 5.1) is similar to the processes presented in literature. However, instead of relying on two classical model checking procedures the intersection between BA and IBA has been redefined

[3]Note that we are considering the three value model checking problem and not the generalized model checking which is a different and more complex problem.

[4]The generalized model checking problem has been discussed for example in [12].

(Section 5.1). The reason behind this choice is to use the new extended intersection to extract the sub-properties related with the different components (boxes). Furthermore, as specified Table 8.6, this choice has no impact on the asymptotic behavior of the model checking algorithm.

8.4.3 Sub-property computation

The sub-property computation problem is similar to other problems already analyzed in literature. This section highlights the main differences and similarities between the proposed approach and related work, such as synthesis and supervisory control. An extended discussion on the related work has been presented in Section 2.4. Note that, as mentioned in [56], “the particular framework in which the problems are considered makes all the difference to the proposed solutions".

In program synthesis, the developer usually wants to compute a model of the system that satisfies a set of properties of interest. For example, in [112], the goal is to compute a model of the system C_2 whose interaction with the environment guarantees that a specific formula is satisfied. Differently, in this thesis, the goal is to compute sub-properties for the unspecified components. In this sense, the addressed problem is more similar to the assumption generation problem presented in [56]. In this work, the authors, given a model of the system $\mathcal{M}$, which contains a set of controllable actions compute an assumption, i.e., an environment executed in concurrency with $\mathcal{M}$ that guarantees the satisfaction of the properties of interest. Again, this is a slightly different problem with respect to the one considered in this thesis. In particular, the approach presented in this thesis is more finer-grained since it tries to “allocate" parts of the assumption over the different components located in the system. Furthermore, in [56] only safety properties are considered and the proposed algorithm is exponential since the error $LTS^{\uparrow}$ obtained as an assumption is converted into a deterministic $LTS^{\uparrow}$. An other work which considers the synthesis problem in the context of finite state machines is [129]. In [129] the authors propose a synthesis techniques that constructs *Modal Transition Systems* (MTSs) from combination of *safety properties* and *scenarios*. Again the problem of updating (synthesizing) the model from properties and scenarios is slightly different to the constraint computation algorithm where the goal is to compute sub-properties.

The computation of the constraint can be interpreted as a *supervisory control* problem [16, 92, 113, 114] in which each box is associated with a set of controllable actions. In supervisory control, given a model of the system $\mathcal{M}$, the set of actions is partitioned in two sets: a set of controllable controllable actions, which are the one the controller can constraint, and a set of hidden actions, which are not visible by the controller. The problem is to synthesize a strategy the controller can employ to modify the behavior of the model $\mathcal{M}$ in a way that satisfies the properties of interest. In this sense, supervisory control is more similar to the assumption generation problem presented in [56].

Cimatti et al., [30] propose an approach to compute contracts while the software is developed. A contract is specified as a tuple $\langle A, G \rangle$, where A is an assumption and G is the guarantee that must be satisfied by the replacement. The idea is independent from the formalism which is used to represent A and G, the only requirement is that A and G have an equivalent trace semantic. The replacement I satisfies the contract if and only if $I \cap [A] \subseteq [G]$, where $[A]$ and $[G]$ represent the set of traces which correspond

to the assumption A and the guarantee G. Withing this setting, our approach can be interpreted as a formalism to compute $[G]$ when no assumption on the behavior of the environment are required. Note that our procedure generate contracts which are in a normal form, i.e., $\overline{[A]} \subseteq [G]$, indeed there are no assumption over the behavior of the environment A which contains all the possible executable traces. Therefore, $\overline{[A]}$ is an empty set of traces.

8.4.4 Replacement checking

The main goal of the replacement checking process is to verify, after a change, only the portion of the state space affected by the change. The idea presented in this thesis is to pre-compute a constraint that will be used after any change for reducing the verification effort. For this reason, problems such as “compositional reasoning", “component substitutability" and “hierarchical model checking" are related to our work.

Compositional reasoning [37] tries to reduce the verification effort by verifying properties on individual components and inferring the properties that hold in the global system without its explicit creation. For example, in the assume-guarantee paradigm [4, 69, 109, 109], if the model M guarantees ϕ, and the model M' guarantees ψ when it is located in an environment that satisfies ϕ, then when M and M' are executed in parallel, they satisfy ψ. By verifying that M guarantees ϕ and M' guarantees ψ when ϕ is satisfied it is possible to infer that the system composed by M and M' satisfies ψ without the generation of the corresponding state space. In some sense, this is the dual problem of the one considered in this thesis, where starting from the global system the developer wants to infer properties that must hold into the single components.

In component substitutability [122], the problem concerns the verification of a system when a component is removed and replaced by a new one. The main idea is to check that the new component preserves the behaviors provided by the hold one. Again, the problem is slightly different from the one considered in this work.

Finally, the model checking of Hierarchical State Machines (HSMs) [6–8] is related to our work since HSMs support iterative refinement in a way which is similar to the IBAs case. However, the verification procedures for HSMs assume that the HSMs is fully specified, i.e., the verification can be performed only at the end of the software development cycle. Conversely, this thesis proposes a verification technique that aims to distribute the verification effort in a more uniform way along the development cycle allowing the verification of models that are iteratively refined.

CHAPTER 9

Conclusion and Future Work

"I am still learning."

Michelangelo di Lodovico Buonarroti Simoni, 1475-1564

The software development process is an iterative activity. The final artifact is obtained through a sequence of development rounds in which new components are defined, different design solutions are analyzed, off-the-shelf components are evaluated. *Incompleteness* is a constant element hailing from this dynamic and evolving environment. The initial solution proposed by the developer usually specifies a wide collection of possibly nonequivalent implementations, i.e., it is incomplete. The incomplete parts are constantly reduced during the design process, until one single implementation is determined [77]. The development of certain functionalities may be simply postponed or the implementation will be provided by a third party, such as in the case of a component-based or a service-based system. When the implementation of a functionality is postponed, the developer usually defines the corresponding behavior in some later stage of the development process, e.g., after exploring alternative solutions and evaluating their trade-offs. There are also cases in which the postponed functionality may become available at *run-time*, as in the case of dynamically adaptive systems. A similar process is performed when the system undergoes future evolution.

Most of the analysis techniques on the market ignore incompleteness. In this setting, the benefit of analysis only appears at the end of a costly process of constructing a comprehensive behavior model, which contains a full description of the system. Conversely, highly explorative, iterative, and incremental model-driven design approaches require the existing formal verification techniques to be profoundly revisited, i.e., they should accomodate *incomplete* models. This thesis proposed a modeling and reasoning framework that supports the designer during the software development process, from

the early stages of the software development cycle, where high level and incomplete models are considered, to the final implementation.

9.1 Summary of the Thesis

This thesis aimed to reduce the gap between the current analysis techniques, in which incompleteness is in general neglected, and the need of techniques to be integrated in the software development process, which is an iterative and incremental activity. In particular,

- We proposed IBAs, a *modeling formalism* that supports incremental development. IBAs extend classical Büchi Automata (BAs) by partitioning the set of states into regular and black box states. Black box states are used to represent components that will be later refined. IBAs allow the specification to describe a collection of models, which are constantly reduced during the design process, until the final model of the system is defined. This answers the Question 1.2.1 regarding how to specify systems which contain parts whose refinement is postponed. A comparison with the existing modeling formalisms that support incompleteness is presented in Chapter 2 and Section 8.4.1.

- We have defined a *refinement relationship* (Definition 4.2.1), which specifies when an IBA $\mathcal{N}$ refines an IBA $\mathcal{M}$. We have formally defined a replacement (Definition 4.2.3), and how this replacement can be plugged into the original IBA (Definition 4.2.8). We have demonstrated that the automaton obtained by plugging the replacement into the original IBA is a refinement (Theorem 4.2.2). This answers the Question 1.2.2 which concerns how it is possible to specify incompleteness and to update the model of the system when the replacements of new components are available.

- We have proposed a *model checking* technique that supports incomplete models. We have defined a semantic of LTL over IBA (Definition 4.3.1) and, starting from this semantic, we have proposed a model checking technique. The model checking technique considers a model of the system specified through an IBA against a property specified as an LTL formula, which is converted into the corresponding BA, or directly as a BA. The model checking technique returns three possible values, true if the property is satisfied, false if it is not, maybe if the satisfaction of the property depends on the still to be refined components. Sections 2.3 and 8.4.2 compare our model checking technique with state of the art procedures. This answers the Question 1.2.3 regarding the use of the model checking techniques in the context of incomplete specifications.

- We have proposed an algorithm that *synthesizes a constraint* for the incomplete parts. The developer may use this algorithm to compute the sub-property related to each black box state of the IBA. The sub-property contains the weakest assumption the developer may satisfy in the refinement activity. We have demonstrated the correctness of the procedure (Theorem 5.2.3). This answers the Question 1.2.5 on how is it possible to compute a constraint (a set of requirements) that the developer must consider in the refinement of incompleteness.

- We have proposed an algorithm to perform the *replacement verification*. The algorithm does not re-verify everything from scratch, i.e., by generating a refinement and checking the refinement against the original requirements, but, conversely, it considers only the replacement against the previously computed constraint. This answers the Question 1.2.4 on how to re-check the system when a new replacement is proposed.
- We have evaluated the proposed approach over two different *case studies* (Chapter 7). For each of them, we described two different refinement scenarios. The first case study concerns the well known mutual exclusion system which has been considered in several works, such as [9, 104]. The second is a real case study which has been described in [139]. The results (see also Section 8.3) demonstrate the advantages of the proposed approach and entail further analysis and evaluations.
- We have discussed the *complexities* of the incomplete model checking, constraint computation and replacement verification algorithms proposed in Chapter 5. We have also compared the complexity of our procedure with existing state of the art procedures (Section 8.4).
- We have analyzed the *scalability of the approach* (Section 8.2). In particular, we have compared the efficiency of the refinement verification (Section 5.1) against the replacement checking (Section 5.3). In absence of a realistic benchmark suite, we have *randomly* generated a set of IBAs and BAs. By controlling the density of transitions, of the accepting and black box states, we have analyzed the performance of the algorithms over different types of models. The evaluation illustrates the configurations (values of the parameters) for which the verification of the replacement outperforms the verification of the refinement.

We believe that our work also provides the following contributions:

- It introduces a sound and complete framework for designing and analyzing systems in a *hierarchical* development setting. In particular, the thesis proposes both a modeling formalism that natively supports the system design and its iterative refinement through a hierarchical decomposition.
- It supports *step-wise refinement*. The developer is encouraged in the description of general functions and, then, in their iterative decomposition until the whole program is fully defined.
- It allows an *early detection of property violations*. The allocation of functions to the sub-components is decided and proven correct at the moment of the decomposition. Instead of being obsessed by providing a complete design of the system, which will then be verified, the developer can decompose the design process in refinement rounds. At each refinement round, a single component (or a single functionality) is defined and checked against the property of interest.
- It encourages a more *conscious development* and *separation of concerns*. The developer before designing a component may compute the corresponding sub-property. The sub-properties specify the set of behaviors that violate and possibly violate the property. An analysis of these behaviors describe what the component should and should not do. Furthermore, when the replacements of different

components (black box states) are designed by various development teams, the constraint analysis allows the detection of possible flaws which are consequence of the interaction among the components.

9.2 Future work

The thesis offers different directions for future works:

- *Case studies*. An interesting line of the research concerns the use of the approach in a larger set of case studies. This can provide additional feedback on the advantages and disadvantages of the approach. Furthermore, the scalability evaluation performed on this thesis has been executed on a set of randomly generated models. There is no guarantee that the randomly generated models reflect possible design decisions of the developer. One can argue, for example, that the transition density is usually lower/higher in real case studies.

- *Extension of the modeling formalism*. As evidenced in Section 8.3, sometimes the developer may want to specify assumption, i.e., an invariant over the value of the proposition inside a black box state. For example, a component $\mathcal{R}$ may not be able to change the value of the proposition a. The introduction of this additional feature would enrich the formalism by supporting a higher separation of concerns. Furthermore, Table 8.5 presented in Section 8.4.1 specifies that, at the current state, IBAs do not support concurrency. An interesting extension concerns the definition of parallel execution of IBAs. This could give the developer more expressive power and would allow the use of IBAs in the modeling of concurrent systems without the manual generation of all the possible interleavings between two IBAs. Finally, other extensions may concern the addition of incompleteness over transitions (as in the case of MTSs) or propositions (as for PKSs), which can increase the expressiveness of the modeling formalism.

- *Extension of the model checking procedure*. As highlighted in Table 8.6 presented in Section 8.4.2, no model checking algorithm has been proposed to check CTL formulae over IBAs. A possible extension concerns the definition of the CTL semantic over IBA, and the development of the corresponding model checking framework.

- *Expression of the sub-property*. As described in Section 5.2, the constraint computation algorithm returns for each black box state a sub-property which is made by an automaton which relates the behavior of the replacement to the one of the original model through incoming and outgoing transitions and their labels. However, sometimes these automata are difficult to be interpreted, since they encode all the possible behaviors which violate and possibly violate the properties of interest. An interesting question is if and how is it possible to express these behaviors in terms of LTL formula (in case augmented with additional operators that specify how the formula is releated with the original model).

- *Selection of the replacement checking procedure*. Section 8.2 compares the performance of the model checking procedure applied to the replacement and the corresponding constraint versus the verification of the refined model against the

original property. The results evidenced that in average the replacement checking is faster than checking the refinement from scratch. However, when the replacement checking is slower, the additional overhead is polynomial. For this reason it may be useful to define appropriate heuristics to dynamically select the model checking procedure to be applied in relation with the values of the parameters described in Table 8.1.

- *Incompleteness in the claim.* The reasoning techniques proposed in this thesis support the developer in the system design when parts of the model are iteratively specified, using a top-down development style. Since both the model and the claim are expressed in terms of automata, an interesting line of research concerns the analysis of the dual case, in which the model is completely specified and the claim contains parts which are unknown. This problem is also known in literature as query checking [20]. As stated in [24], query checking offers two main advantages: *a*) understanding: the user can check the model $\mathcal{M}$ against partially defined properties getting a better insight of what the model does; *b*) sanity check: the user can compare the expectations of the incomplete properties evaluated over the model and the results of the query checking procedure. An interesting question is how the technique can be adapted to synthesize formulae and how can be compared with existing state of the art techniques.

- *Integrate theorem proving techniques*. Theorem proving, or deductive verification, is an alternative approach to model checking. Instead of looking at a counterexample, in theorem proving the goal is to derive that the property is a theorem of the checked model, i.e., given the statements of the model, it is demonstrated that the property holds, by performing the steps of the proof. The main benefit of deductive verification is the possibility to actually explain how and why the system meets its specification. The question is if and how is it possible to integrate theorem proving techniques in an incomplete context, and in particular when IBAs and LTL formulae are considered.

- *Improving the automated support.* The tool has been developed as a proof of concepts and needs optimization. The tool offers a good base-point that allows checking how the procedure can be used in practice but has several limitation:

  - The tool *has no* GUI. Encoding and designing the model of the system through XML files is sometime cumbersome, time consuming and error prone. Conversely, a graphical user interface will give the user a better support in the development activity.
  - The current implementation is not *highly optimized*. For example, the data structures used in the refinement and replacement verification are not high efficient and can be replaced by others which provide better performances. Furthermore, the model checking algorithm can be re-implemented exploiting symbolic techniques and relying on state of the art model checking tools, such as NuSMV [27].

Bibliography

[1] Cliche command-line shell. https://code.google.com/p/cliche/. Accessed: 2013-03-26.

[2] Jaxb. `https://jaxb.java.net/`. Accessed: 2015-02-10.

[3] Ltl2ba4j - java bridge to ltl2ba. `http://www.sable.mcgill.ca/~ebodde/rv/ltl2ba4j/`. Accessed: 2015-02-10.

[4] Rajeev Alur and ThomasA. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.

[5] Rajeev Alur, Sampath Kannan, and Mihalis Yannakakis. Communicating hierarchical state machines. In *Automata, Languages and Programming*, pages 169–178. Springer, 1999.

[6] Rajeev Alur and Mihalis Yannakakis. Model checking of hierarchical state machines. *SIGSOFT Softw. Eng. Notes*, 23(6):175–188, November 1998.

[7] Rajeev Alur and Mihalis Yannakakis. Model checking of hierarchical state machines. In *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '98/FSE-6, pages 175–188, New York, NY, USA, 1998. ACM.

[8] Rajeev Alur and Mihalis Yannakakis. Model checking of hierarchical state machines. *ACM Trans. Program. Lang. Syst.*, 23(3):273–303, May 2001.

[9] Christel Baier, Joost-Pieter Katoen, et al. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.

[10] Yitzhak Brave. Control of discrete event systems modeled as hierarchical state machines. *Automatic Control, IEEE Transactions on*, 38(12):1803–1819, 1993.

[11] Glenn Bruns and Patrice Godefroid. Model checking partial state spaces with 3-valued temporal logics. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 274–287. Springer Berlin Heidelberg, 1999.

[12] Glenn Bruns and Patrice Godefroid. Generalized model checking: Reasoning about partial state spaces. In *CONCUR 2000-Concurrency Theory*, pages 168–182. Springer, 2000.

[13] Glenn Bruns and Patrice Godefroid. Generalized model checking: Reasoning about partial state spaces. In Catuscia Palamidessi, editor, *CONCUR 2000 - Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 168–182. Springer Berlin Heidelberg, 2000.

[14] J Richard Büchi. On a decision method in restricted second order arithmetic. In *Proc. Internat. Congr. Logic, Method. and Philos. Sci*, pages 1–12, 1960.

[15] J Richard Buchi and Lawrence H Landweber. *Solving sequential conditions by finite-state strategies*. Springer, 1990.

[16] Christos G Cassandras et al. *Introduction to discrete event systems*. Springer Science & Business Media, 2008.

[17] Sagar Chaki, Edmund Clarke, Alex Groce, Joël Ouaknine, Ofer Strichman, and Karen Yorav. Efficient verification of sequential and concurrent c programs. *Formal Methods in System Design*, 25(2-3):129–166, 2004.

[18] Sagar Chaki, Edmund Clarke, Natasha Sharygina, and Nishant Sinha. Verification of evolving software via component substitutability analysis. *Formal Methods in System Design*, 32(3):235–266, 2008.

[19] Sagar Chaki, James Ivers, Natasha Sharygina, and Kurt Wallnau. The comfort reasoning framework. In *Computer Aided Verification*, pages 164–169. Springer, 2005.

[20] William Chan. Temporal-logic queries. In *Computer Aided Verification*, pages 450–463. Springer, 2000.

[21] Marsha Chechik, Greg Brunet, Dario Fischbein, Sebastian Uchitel, Ed Brinksma, David Harel, Angelika Mader, Perdita Stevens, and Roel Wieringa. Partial behavioural models for requirements and early design. *MMOSS*, 6351, 2006.

[22] Marsha Chechik, Benet Devereux, Steve Easterbrook, and Arie Gurfinkel. Multi-valued symbolic model-checking. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(4):371–408, 2003.

[23] Marsha Chechik, Steve Easterbrook, and Victor Petrovykh. Model-checking over multi-valued logics. In *FME 2001: Formal Methods for Increasing Software Productivity*, pages 72–98. Springer, 2001.

[24] Hana Chockler, Arie Gurfinkel, and Ofer Strichman. Variants of ltl query checking. In *Hardware and Software: Verification and Testing*, pages 76–92. Springer, 2011.

[25] Yaacov Choueka. Theories of automata on ω-tapes: A simplified approach. *J. Comput. Syst. Sci.*, 8(2):117–141, April 1974.

[26] Yaacov Choueka. Theories of automata on< i> ω</i>-tapes: A simplified approach. *Journal of computer and system sciences*, 8(2):117–141, 1974.

[27] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, pages 359–364. Springer, 2002.

[28] Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147(1):35–84, 2003.

[29] Alessandro Cimatti and Stefano Tonetta. A property-based proof system for contract-based design. In *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, pages 21–28. IEEE, 2012.

[30] Alessandro Cimatti and Stefano Tonetta. Contracts-refinement proof system for component-based embedded systems. *Science of Computer Programming*, 97:333–348, 2015.

[31] Davide Ciucci and Didier Dubois. Three-valued logics for incomplete information and epistemic logic. In *Logics in Artificial Intelligence*, pages 147–159. Springer, 2012.

[32] Edmund M Clarke and E Allen Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic*. Springer, 1982.

[33] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[34] JamiesonM. Cobleigh, Dimitra Giannakopoulou, and CorinaS. PAsAreanu. Learning assumptions for compositional verification. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer Berlin Heidelberg, 2003.

[35] James R Cordy. The txl source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[36] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In EdmundM. Clarke and RobertP. Kurshan, editors, *Computer-Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 233–242. Springer Berlin Heidelberg, 1991.

[37] W-P de Roever. *Concurrency Verification: Introduction to Compositional and Non-compositional Methods*, volume 54. Cambridge University Press, 2001.

[38] Martin De Wulf, Laurent Doyen, Thomas A Henzinger, and J-F Raskin. Antichains: A new algorithm for checking universality of finite automata. In *Computer Aided Verification*, pages 17–30. Springer, 2006.

[39] Nicolas D'Ippolito, Victor Braberman, Nir Piterman, and Sebastian Uchitel. Synthesis of live behaviour models for fallible domains. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 211–220. IEEE, 2011.

[40] Nicolas D'ippolito, Victor Braberman, Nir Piterman, and Sebastián Uchitel. Synthesizing nonanomalous event-based controllers for liveness goals. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(1):9, 2013.

[41] Nicolas D'Ippolito, Dario Fischbein, Marsha Chechik, and Sebastián Uchitel. Mtsa: The modal transition system analyser. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 475–476. IEEE Computer Society, 2008.

[42] Calvin C Elgot. Decision problems of finite automata design and related arithmetics. *Transactions of the American Mathematical Society*, pages 21–51, 1961.

[43] E Allen Emerson and Chin-Laung Lei. Temporal reasoning under generalized fairness constraints. In *STACS 86*, pages 21–36. Springer, 1986.

[44] Michalis Famelis, Rick Salay, and Marsha Chechik. Partial models: Towards modeling and reasoning with uncertainty. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 573–583, Piscataway, NJ, USA, 2012. IEEE Press.

[45] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. Run-time efficient probabilistic model checking. In *Proceedings of the 33rd international conference on software engineering*, pages 341–350. ACM, 2011.

[46] Dario Fischbein. *Foundations for behavioural model elaboration using modal transition systems*. PhD thesis, Imperial College London, 2012.

[47] Dario Fischbein, Victor Braberman, and Sebastián Uchitel. A sound observational semantics for modal transition systems. In *Theoretical Aspects of Computing-ICTAC 2009*, pages 215–230. Springer, 2009.

[48] Dario Fischbein, Nicolas D'Ippolito, Greg Brunet, Marsha Chechik, and Sebastian Uchitel. Weak alphabet merging of partial behavior models. *ACM Trans. Softw. Eng. Methodol.*, 21(2):9:1–9:47, March 2012.

[49] Dario Fischbein and Sebastian Uchitel. On correct and complete strong merging of partial behaviour models. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 297–307. ACM, 2008.

[50] Martin Fowler and Jim Highsmith. The agile manifesto. *Software Development*, 9(8):28–35, 2001.

[51] Benoit Gaudin and Hervé Marchand. Safety control of hierarchical synchronous discrete event systems: A state-based approach. In *Intelligent Control, 2005. Proceedings of the 2005 IEEE International Symposium on, Mediterrean Conference on Control and Automation*, pages 889–895. IEEE, 2005.

[52] Rob Gerth, Doron Peled, Moshe Y. Vardi, R. Gerth, Den Dolech Eindhoven, D. Peled, M. Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *In Protocol Specification Testing and Verification*, pages 3–18. Chapman Hall, 1995.

[53] Rob Gerth, Doron Peled, Moshe Y Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *International Symposium on Protocol Specification, Testing and Verification*. IFIP, 1995.

[54] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice Hall PTR, 2002.

[55] Dimitra Giannakopoulou and Jeff Magee. Fluent model checking for event-based systems. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 257–266. ACM, 2003.

[56] Dimitra Giannakopoulou, Corina S Păsăreanu, and Howard Barringer. Assumption generation for software component verification. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 3–12. IEEE, 2002.

[57] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In *Computer aided verification*, pages 72–83. Springer, 1997.

[58] Jonathan L Gross and Jay Yellen. *Handbook of graph theory*. CRC press, 2004.

[59] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Trans. Program. Lang. Syst.*, 16(3):843–871, May 1994.

[60] Arie Gurfinkel and Marsha Chechik. Temporal logic query checking through multi-valued model checking. Technical report, Citeseer, 2002.

[61] Arie Gurfinkel and Marsha Chechik. Why waste a perfectly good abstraction? In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 212–226. Springer, 2006.

[62] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[63] Matthew Hennessy and Gordon Plotkin. Finite conjunctive nondeterminism. In *Concurrency and Nets*, pages 233–244. Springer, 1987.

[64] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[65] Gerard J Holzmann. *The SPIN model checker: Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.

[66] Michael Huth. Model checking modal transition systems using kripke structures. In Agostino Cortesi, editor, *Verification, Model Checking, and Abstract Interpretation*, volume 2294 of *Lecture Notes in Computer Science*, pages 302–316. Springer Berlin Heidelberg, 2002.

[67] Michael Huth, Radha Jagadeesan, and David Schmidt. Modal transition systems: A foundation for three-valued program analysis. In *Programming Languages and Systems*, pages 155–169. Springer, 2001.

[68] Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[69] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):596–619, 1983.

[70] Bernhard Josko. Verifying the correctness of aadl modules using model checking. In *Stepwise Refinement of Distributed Systems Models, Formalisms, Correctness*, pages 386–400. Springer, 1990.

[71] Stephen Cole Kleene. Introduction to metamathematics. 1952.

[72] Kai Koskimies and Erkki Mäkinen. Automatic synthesis of state machines from trace diagrams. *Software: Practice and Experience*, 24(7):643–658, 1994.

[73] Orna Kupfermant and Moshe Y Vardit. Synthesis with incomplete informatio. In *Advances in Temporal Logic*, pages 109–127. Springer, 2000.

[74] Leslie Lamport. What good is temporal logic? In *IFIP congress*, volume 83, pages 657–668, 1983.

[75] Kim G Larsen. *Context-dependent bisimulation between processes.* PhD thesis, University of Edinburgh, 1986.

[76] Kim G. Larsen, Bernhard Steffen, and Carsten Weise. A constraint oriented proof methodology based on modal transition systems. In *In BRICS Notes*, pages 17–40. Springer-Verlag, 1995.

[77] Kim G Larsen and Bent Thomsen. A modal process logic. In *Logic in Computer Science, 1988. LICS'88., Proceedings of the Third Annual Symposium on*, pages 203–210. IEEE, 1988.

[78] Kim Guldstrand Larsen. The expressive power of implicit specifications. In *Automata, Languages and Programming*, pages 204–216. Springer, 1991.

[79] Christoph Legat, Jens Folmer, and Birgit Vogel-Heuser. Evolution in industrial plant automation: A case study. In *Industrial Electronics Society, IECON 2013-39th Annual Conference of the IEEE*, pages 4386–4391. IEEE, 2013.

[80] Emmanuel Letier, Jeff Kramer, Jeff Magee, and Sebastian Uchitel. Deriving event-based transition systems from goal-oriented requirements models. *Automated Software Engineering*, 15(2):175–206, 2008.

[81] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 97–107. ACM, 1985.

[82] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '85, pages 97–107, New York, NY, USA, 1985. ACM.

[83] Pericles Loucopoulos and Vassilios Karakostas. *System requirements engineering*. McGraw-Hill, Inc., 1995.

[84] Jeff Magee and Jeff Kramer. *State models and java programs*. wiley, 1999.

[85] Zohar Manna and Pierre Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(1):68–93, 1984.

[86] H. Marchand and B. Gaudin. Supervisory control problems of hierarchical finite state machines. In *Decision and Control, 2002, Proceedings of the 41st IEEE Conference on*, volume 2, pages 1199–1204 vol.2, Dec 2002.

[87] George H Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[88] Bertrand Meyer. Applying'design by contract'. *Computer*, 25(10):40–51, 1992.

[89] Robin Milner. A calculus of communicating systems. 1980.

[90] Robin Milner. Calculi for synchrony and asynchrony. *Theoretical computer science*, 25(3):267–310, 1983.

[91] Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., 1989.

[92] Sajed Miremadi, Bengt Lennartson, and K Akesson. Bdd-based supervisory control on extended finite automata. In *Automation Science and Engineering (CASE), 2011 IEEE Conference on*, pages 25–31. IEEE, 2011.

[93] Edward F Moore. Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153, 1956.

[94] Barak Naveh. Jgrapht a free java graph library, 2011.

[95] Tobias Nopper and Christoph Scholl. Approximate symbolic model checking for incomplete designs. In *Formal Methods in Computer-Aided Design*, pages 290–305. Springer, 2004.

[96] Tobias Nopper and Christoph Scholl. Symbolic model checking for incomplete designs with flexible modeling of unknowns. *Computers, IEEE Transactions on*, 62(6):1234–1254, 2013.

[97] David Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer Berlin Heidelberg, 1981.

[98] David Park. *Concurrency and automata on infinite sequences*. Springer, 1981.

[99] Karl Pearson. Note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London*, pages 240–242, 1895.

[100] Doron Peled. *Software reliability methods*. Springer, 2001.

[101] Doron Peled. From verification to synthesis. In *Dependable Software Systems Engineering*, pages 204–223. 2015.

[102] Doron Peled and Thomas Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters*, 63(5):243–246, 1997.

[103] Doron Peled, Thomas Wilke, and Pierre Wolper. An algorithmic approach for checking closure properties of ω-regular languages. In *CONCUR'96: Concurrency Theory*, pages 596–610. Springer, 1996.

[104] Doron Peled and Lenore Zuck. From model checking to a temporal proof. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 1–14. Springer-Verlag New York, Inc., 2001.

[105] Doron A Peled. *Software reliability methods*. Springer Science & Business Media, 2013.

[106] Nir Piterman. From nondeterministic buchi and streett automata to deterministic parity automata. In *Logic in Computer Science, 2006 21st Annual IEEE Symposium on*, pages 255–264. IEEE.

[107] Gordon D Plotkin. A structural approach to operational semantics. 1981.

[108] Amir Pneuli and Roni Rosner. Distributed reactive systems are hard to synthesize. In *Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on*, pages 746–757. IEEE, 1990.

[109] A. Pnueli. Logics and models of concurrent systems. chapter In Transition from Global to Modular Temporal Reasoning About Programs, pages 123–144. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[110] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 179–190, New York, NY, USA, 1989. ACM.

[111] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[112] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190. ACM, 1989.

[113] Peter J Ramadge and W Murray Wonham. Supervisory control of a class of discrete event processes. *SIAM journal on control and optimization*, 25(1):206–230, 1987.

[114] Peter JG Ramadge and W Murray Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[115] Andrew William Roscoe, Charles AR Hoare, and Richard Bird. *The theory and practice of concurrency*, volume 169. Prentice Hall Englewood Cliffs, 1998.

[116] Karen Rudie and Murray W Wonham. Think globally, act locally: Decentralized supervisory control. *Automatic Control, IEEE Transactions on*, 37(11):1692–1708, 1992.

[117] Shmuel Safra. On the complexity of omega-automata. In *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*, pages 319–327. IEEE Computer Society, 1988.

[118] Rick Salay, Marsha Chechik, and Jan Gorzny. Towards a methodology for verifying partial model refinements. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 938–945. IEEE, 2012.

[119] Antonella Santone, Gigliola Vaglini, and Maria Luisa Villani. Incremental construction of systems: An efficient characterization of the lacking sub-system. *Science of Computer Programming*, 78(9):1346 – 1367, 2013.

[120] Amir Molzam Sharifloo and Paola Spoletini. Lover: light-weight formal verification of adaptive systems at run time. In *Formal Aspects of Component Software*, pages 170–187. Springer, 2013.

[121] Natasha Sharygina, Sagar Chaki, Edmund Clarke, and Nishant Sinha. Dynamic component substitutability analysis. In *FM 2005: Formal Methods*, pages 512–528. Springer, 2005.

[122] Sagar Chaki Natasha Sharygina and Nishant Sinha. Verification of evolving software. *SAVCBS 2004 Specification and Verification of Component-Based Systems*, page 55, 2004.

[123] Sharon Shoham and Orna Grumberg. Monotonic abstraction-refinement for ctl. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 546–560. Springer, 2004.

[124] Gil Shurek and Orna Grumberg. The modular framework of computer-aided verification. In *Computer-Aided Verification*, pages 214–223. Springer, 1991.

[125] Markus Skoldstam, Knut Akesson, and Martin Fabian. Modeling of discrete event systems using finite automata with variables. In *Decision and Control, 2007 46th IEEE Conference on*, pages 3387–3392. IEEE, 2007.

[126] Deian Tabakov and Moshe Y Vardi. Experimental evaluation of classical automata constructions. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 396–411. Springer, 2005.

[127] Deian Tabakov and Moshe Y Vardi. Model checking buechi specifications. In *LATA*, pages 565–576. Citeseer, 2007.

[128] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[129] S. Uchitel, G. Brunet, and M. Chechik. Synthesis of partial behavior models from properties and scenarios. *Software Engineering, IEEE Transactions on*, 35(3):384–406, May 2009.

[130] Sebastian Uchitel, Dalal Alrajeh, Shoham Ben-David, Victor Braberman, Marsha Chechik, Guido De Caso, Nicolas D'Ippolito, Dario Fischbein, Diego Garbervetsky, Jeff Kramer, et al. Supporting incremental behaviour model elaboration. *Computer Science-Research and Development*, 28(4):279–293, 2013.

[131] Sebastian Uchitel and Marsha Chechik. Merging partial behavioural models. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 43–52. ACM, 2004.

[132] Sebastian Uchitel and Marsha Chechik. Merging partial behavioural models. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 43–52, New York, NY, USA, 2004. ACM.

[133] Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Synthesis of behavioral models from scenarios. *Software Engineering, IEEE Transactions on*, 29(2):99–115, 2003.

[134] Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Incremental elaboration of scenario-based specifications and behavior models using implied scenarios. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 13(1):37–85, 2004.

[135] Axel Van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 249–262. IEEE, 2001.

[136] Axel Van Lamsweerde et al. Requirements engineering: from system goals to uml models to software specifications. 2009.

[137] Moshe Y Vardi. Why is modal logic so robustly decidable? *Descriptive complexity and finite models*, 31:149–184, 1996.

[138] Moshe Y Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*. IEEE Computer Society, 1986.

[139] Birgit Vogel-Heuser, Christoph Legat, Jens Folmer, and Stefan Feldmann. Researching evolution in industrial plant automation: Scenarios and documentation of the pick and place unit. Technical report, Technical Report TUM-AIS-TR-01-14-02, Institute of Automation and Information Systems, Technische Universität München, 2014.

[140] Richard J Waldinger and Richard CT Lee. Prow: A step toward automatic program writing. In *Proceedings of the 1st international joint conference on Artificial intelligence*, pages 241–252. Morgan Kaufmann Publishers Inc., 1969.

[141] Ou Wei, Arie Gurfinkel, and Marsha Chechik. Mixed transition systems revisited. In *Verification, Model Checking, and Abstract Interpretation*, pages 349–365. Springer, 2009.

[142] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[143] Kai C Wong and W Murray Wonham. Hierarchical control of discrete-event systems. *Discrete Event Dynamic Systems*, 6(3):241–273, 1996.

[144] W Murray Wonham and Peter J Ramadge. Modular supervisory control of discrete-event systems. *Mathematics of control, Signals and Systems*, 1(1):13–30, 1988.

[145] T-S Yoo and Stéphane Lafortune. A general architecture for decentralized supervisory control of discrete-event systems. *Discrete Event Dynamic Systems*, 12(3):335–377, 2002.

[146] Michal Young. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

APPENDIX A

Tool documentation

The CHIA tool is an interactive command line framework which allows to iteratively refine and check models.

A.1 Installing CHIA

The CHIA tool can be found at *https://github.com/claudiomenghi/CHIA/*. To install CHIA

- download the `CHIA.jar` file
- run `java -jar CHIA.jar`

By running the `java -jar CHIA.jar` command the shell of CHIA is executed. The user interacts with the tool using a set of commands.

A.2 Commands

The CHIA tool can be executed in two modalities: automata and replacement checking.

- `exit`: exits the CHIA framework.
- `automata` (`aut`): enters the automata mode.
- `replacement` (`rep`): enters the replacement mode.

Automata mode commands
The automata commands can be used in the automata mode when a Model (IBA) is considered against the constraint property.

- `lm modelFilePath`: is used to load the model from an XML file. The XML file must match the IBA.xsd. It requires the parameter `modelFilePath`, i.e., the path of the file that contains the model to be checked.
- `dispm`: is used to display the model into the console.
- `lp propertyFilePath`: is used to load the property from an XML file. The XML file must mach the BA.xsd. It requires the parameter `propertyFilePath`, i.e., the path of the file that contains the property to be checked.
- `dispp`: is used to display the property into the console.

- `lpLTL LTLFormula`: it is used to load the property from an LTL formula. `LTLFormula` is the LTL formula that represents the property to be checked. CHIA uses `LTL2BA4J` [3] to translate LTL formulae into Büchi automaton. The LTL formula can be created starting from a set of propositional symbols, i.e., $true$, $false$ any lowercase string, a set of boolean operators, i.e., ! (negation), -> (implication), <-> (equivalence), $\wedge$ (and), v (or), and a set of temporal operators, [] (always) <> (eventually), U (until), R (realease) (Spin syntax : V), X (next).
- `ck`: is used to check the model against the specified formula. Before running the model checking procedure it is necessary to load the model and the property to be considered.
- `cc`: computes the constraint corresponding to the model and the specified property.
- `sc constraintFilePath`: saves the constraint in an XML file. The path of the file where the constraint must be saved is specified by the parameter `constraintFilePath`.
- `dispc`: is used to display the constraint into the console.

Replacement mode commands
The automata commands can be used in the replacement mode when a Replacement (BA) is considered against the constraint or a specific sub-property.

- `lc constraintFilePath`: is used to load the constraint from an XML file. The XML file must mach the `Constraint.xsd` file. The parameter `constraintFilePath` specifies the path of the file that contains the constraint to be considered.
- `dispc`: is used to display the constraint into the console.
- `lr replacementFilePath`: is used to load the replacement from an XML file. The XML file must mach the `Replacement.xsd`. The parameter `replacementFilePath` specifies the path of the file that contains the replacement to be considered.
- `dispr`: is used to display the replacement into the console.
- `ck`: is used to check the replacement against the corresponding sub-property.

A.3 Using CHIA via Maven

The CHIA framework is published at: `https://github.com/claudiomenghi/CHIA/`. Before loading the different modules it is necessary to add the repository dependency to the specified MAVEN project as follows:

```
<repositories>
    <repository>
        <id>IncompleteAutomataBasedModelChecking−mvn−repo</id>
        <url>https://raw.github.com/claudiomenghi/CHIA/mvn−repo</url>
         <snapshots>
            <enabled>true</enabled>
            <updatePolicy>always</updatePolicy>
        </snapshots>
    </repository>
</repositories>
```

The `CHIAAutomata` module described in Section 6.2 can be included in the project by adding the following dependency:

```
<dependency>
    <groupId>it.polimi.chia</groupId>
    <artifactId>CHIAAutomata</artifactId>
    <version>0.0.1−SNAPSHOT</version>
</dependency>
```

The `CHIAAutomataIO` module described in Section 6.2.1 can be included in the project by adding the following dependency:

```
<dependency>
    <groupId>it.polimi.chia</groupId>
    <artifactId>CHIAAutomataIO</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

The `CHIAChecker` module described in Section 6.3 can be included in the project by adding the following dependency:

```
<dependency>
    <groupId>it.polimi.chia</groupId>
    <artifactId>CHIAChecker</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

The `CHIAConstraint` module described in Section 6.4 can be included in the project by adding the following dependency:

```
<dependency>
    <groupId>it.polimi.chia</groupId>
    <artifactId>CHIAConstraint</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

The `CHIAConstraintIO` module described in Section 6.4.1 can be included in the project by adding the following dependency:

```
<dependency>
    <groupId>it.polimi.chia</groupId>
    <artifactId>CHIAConstraintIO</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

The `CHIAContraintComputation` module described in Section 6.5 can be included in the project by adding the following dependency:

```
<dependency>
    <groupId>it.polimi.chia</groupId>
    <artifactId>CHIAConstraintComputation</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

The `CHIAReplacementChecker` module described in Section 6.6 can be included in the project by adding the following dependency:

```
<dependency>
    <groupId>it.polimi.chia</groupId>
    <artifactId>CHIAReplacementChecker</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

List of Figures

List of Tables

Acronyms

B

BA	Büchi automaton. 11, 22, 111, 200

C

CCS	Calculus of Communicating Systems. 34, 200
CHIA	CHecher for Incomplete Automata. 189–191, 200
CTL	Computation Tree Logic. 20, 22, 32, 200

D

DFA	Deterministic Finite State Automaton. 28, 200
DFS	Dept First Search. 116, 200
DI	Detected Input. 200
DO	Detected Output. 200
DTD	Document Type Definition. 200

E

ENF	Existential Normal Form. 200

F

FLTL	Fluent Linear Time Temporal Logic. 25, 200
FOL	First Order Logic. 33, 200
FSA	Finite State Automaton. 22, 36, 200
FSM	Finite State Machine. 14, 18, 19, 32, 200

G

GKMTS	Generalized Kripke Modal Transition Systems. 15, 200

H

HSM	Hierarchical State Machine. 16, 200

I

IBA	Incomplete Büchi automaton. 11, 51, 111, 121, 200
IFSA	Incomplete Finite State Automaton. 48, 200
IKS	Incomplete Kripke Structure. 200

K

KMTS	Kripke Modal Transition System. 14, 15, 200
KS	Kripke Structure. 15, 200

L

LTL	Linear Time Temporal Logic. 11, 39, 126, 190, 200
LTS	Labelled transition systems. 14, 22, 25, 27, 28, 200
LTSA	TO DO... 200

M

MCCS	Modal CCS. 200
MPS	Modal Process Logic. 200
MTS	Modal Transition System. 14, 15, 17, 18, 25, 33, 178, 200
MTSA	The Modal Transition System Analyzer. 200

P

PKS	Partial Kripke Structure. 18, 200
PLTS	Partial labelled transition systems. 200
PML	Propositional Modal Logic. 200
PPU	Pick and Place Unit. 137, 138, 194, 200

S

SCC	Strongly Connected Component. 44, 200

W

WP	Working Piece. 200

X

XML	EXtensible Markup Language. 115, 118, 200
XSD	XML Schema Definition. 200

Glossary

abstraction depending on the property to be verified, the systems are reduced by suppressing details that are not relevant in the verification. The idea is that given a property to be verified, abstraction techniques reduces the program state space and generates *only* a smaller set of states that preserve the relevant behaviors of the system. Abstraction techniques are usually performed in an informal manual manner and require considerable expertise. Predicate abstraction [61] is one of the most used and maps concrete data types into abstract data types through predicates on concrete data. 200

accepting state are a specific type of states used to indicate that some operation or task has been completed. 38, 200

behavioral model describe a system as a set of interacting components where each component is modeled as a state machine interacting with other components through shared events. 200

bisimulation is a binary relation between state transition systems, associating systems that behave in the same way in the sense that one system simulates the other and vice-versa. The complexity of checking bisimulation is linear in the size of the models [91, 98]. 18, 200

calculus of Communicating Systems is a process algebra where the developer defines for each state of the system the set of actions that ca be performed (or observed) [89]. 200

compositionality is a property that allows the decomposition of the problem of correctness for a combined system, into *simpler* problems which refer to its subsystems. 200

counterexample is behavior of the system that causes the failure. 200

design process is a sequence of refinement steps that reduce the number of possible implementations of the system. 200

dynamic system a system is dynamic if the output depends on the current and on the *past* values of the input [16]. 200

first order logic extends propositional logic with predicates and quantification. 33, 200

functional requirement specifies a behavior the system is expected to deliver, such as “after a message is sent it is finally delivered" and “two processes cannot access a critical section together" [135]. 2, 200

implementation a model of the system described through a formalism that supports the specification of incomplete parts, such as IBA or MTS, is called implementation if and only if the model does not contains unspecified parts, i.e., behaviors still to be defined. 18, 200

Java Path Finder is a framework to verify programs written in Java. 200

Labeled Transition System Analyzer is a prototype tool available at that supports the specification and analysis of LTS models. 26, 200

liveness is a property that specifies a program eventually enters a desirable state. 200

modal CCS extends CCS with modalities, e.g., may and must.. 200

modal process logic extends process logic with modalities. 200

Modal Transition System Analyser is a tool that supports the construction, analysis and elaboration of Modal Transition System (MTS). The tool is available at `http://sourceforge.net/projects/mtsa/`. 25, 200

model checking is an automatic technique usually performed on a model which abstracts the behavior of the real system and it usually reduces the risk of implementing a flawed design. Given a model of the system $\mathcal{M}$ and a formal property ϕ, the model checking tools exhaustively analyze the state space of $\mathcal{M}$ to check whether all of the system behaviors satisfy ϕ [9]. 2, 200

non functional requirement specifies the software characteristics, such as performance, availability, usability, energy consumption, and costs [83, 136]. 2, 200

partially labeled transition system extends LTS to capture what are the aspects of the system that are still undefined, i.e., they allow to model in each state the set of actions that *may* occur. 200

partially synchronous parallel composition The two automaton composed in parallel synchronize on shared actions but proceed independently on local actions [115]. 200

program synthesis In program synthesis, a specification is transformed into a system that is guaranteed to satisfy the specification. 200

refinement concerns the elaboration of a (incomplete) partial description into a more comprehensive one in which additional detailed are specified. 200

reliability the ability of a system or component to function under stated conditions for a specified period of time. 200

robustness is the ability of a computer system to cope with errors during execution. 200

safety specifies the system never enters an undesirable state. 200

simulation is a relation between state transition systems associating systems which behave in the same way [63] [75]. 18, 200

List of Theorems

List of Theorems