Chapter 4

Abstraction Refinement

Abstraction refinement [CGJ$^+$03, HJMM04] is a promising direction of research to overcome the challenges of the state explosion problem and infinite state model checking, while preserving correctness of verification results. The general abstraction refinement paradigm [CGJ$^+$03] consists of three steps: (1) generate the initial abstraction, (2) model check the abstract system, and, if required, (3) refine the abstraction, and repeat.

The general idea of *abstraction* is to reduce the system complexity, by removing information which is considered irrelevant for the verification of a particular property, and therefore can be safely removed from the system. The obvious problem is how to determine which information is relevant: if the abstraction is too fine—i.e., removes too little information—verification still suffers from the state explosion problem. If, on the other hand, the abstraction is too coarse—i.e., removes too much information—verification suffers from too much information loss,[1] and the results are likely to be wrong. Consequently, abstraction techniques are distinguished based on how they deal with the information loss [CGJ$^+$03].

Over-approximation techniques (also called *conservative techniques*), for example predicate abstraction [GS97], enrich the system behaviour by releasing constraints, such that correctness of the abstract system implies correctness of the concrete system. Over-approximation techniques admit *false negatives* (also called *spurious counterexamples*), i.e., a system behaviour which violates the property in the abstract system, but is not reproducible on the original (concrete) system. *Under-approximation techniques*, on the other hand, constrain the system behaviour by removing irrelevant parts, such that a property violation in the abstract system implies a property violation in the concrete system. Under-approximation techniques admit *false positives*, i.e., a system behaviour which satisfies the property in the abstract system, but violates it in the concrete system. In this work, we deal with

[1]By definition, abstraction *always* means information loss. Yet, with respect to the property to be verified, some information is irrelevant, and can be safely removed.

over-approximation techniques. For a more detailed description of abstraction techniques, see for example [CGP99, CGJ$^+$03].

In the context of abstraction, the general idea of *refinement* can be described as "undo part of the abstraction". If a counterexample to the property under test has been detected in the abstract system (using over-approximation techniques, for under-approximation, the reasoning is different), it needs to be checked whether the counterexample comprises a violation of the property, or whether it is spurious. In the former case, the counterexample to the property is reproducible in the original system, which means the property does not hold in the original system. In the latter case, the counterexample is not reproducible in the original system, which means the spurious counterexample is due to wrong (too coarse) abstraction, and the abstract system needs to be refined. The refinement ideally takes into account the reason why the counterexample has been feasible in the abstract system, in order to prevent this erroneous behaviour in further verification steps.

The remainder of this Chapter is organised as follows: in Section 4.1, we describe our uniform abstraction methodology *abstraction by merging omission*. The methodology is flexible to operate on different system types, since it is defined based on syntactical categories of variables, and takes the constituents of the system under consideration that are to be abstracted as parameters. In Section 4.2, we explain how to translate a counterexample (to the property under test) obtained from the abstract system back into the concrete system. Section 4.3 gives a brief overview of Craig interpolation [Cra57], discusses the expressiveness of the generated interpolants, and how they can be used to derive information about the cause of a spurious counterexample. In the end, we show how to syntactically reorder the input problem (without changing the semantics) to increase the expressiveness of the interpolants. In Section 4.4, we show how to refine the abstraction in case it has turned out to be too coarse, and discuss heuristics on the application of different refinement options. We conclude the Chapter in Section 4.5.

4.1 Abstraction by Merging Omission

In this section, we present a simple and fast, but nevertheless powerful, uniform abstraction technique specifically tailored to work on logical formulas: *abstraction by merging omission (MO)* [KP07, Kem11]. By removing constraints which are considered irrelevant to a particular (safety) property, MO yields an over-approximation.

The basic idea of MO is to reduce the system complexity of a real-time system $\mathfrak{S}$, $\mathfrak{S}\in\{\mathfrak{A}, \mathfrak{T}\}$ (i.e., $\mathfrak{S}$ is a TA or a TCA, for discussion of abstraction of TNA, please refer to Section 6.1), by decreasing the number of symbols in the formula representation $\varphi(\mathfrak{S})$, while retaining as much information as possible about the transition characteristics (the abstract formula is weaker than $\varphi(\mathfrak{S})$, though). MO is defined for formulas in negation normal form (NNF),[2] to which $\varphi(\mathfrak{S})$ can be easily

[2] A formula is defined to be in NNF if negation only appears in front of literals, and $\{\neg, \vee, \wedge\}$ are the only allowed Boolean connectives. In propositional logic, every formula can be transformed into an equivalent formula in NNF, by (1) replacing implications and equivalences by their definitions (i.e., using only $\{\neg, \vee, \wedge\}$), (2) using De Morgan's laws to push negation inside, and (3) eliminating double negations.

transformed. MO uniformly works on the different syntactical categories contained in $\varphi(\mathfrak{S})$: it merges Boolean variables, by mapping them to the same image according to a *map of merging* γ, and it removes rational variables and arithmetic constraints according to a *set of omission* $\mathcal{O}$.

The intended idea of the set of omission $\mathcal{O}$ is to contain constraints and parameters whose *removal* from $\varphi(\mathfrak{S})$ enriches the behaviour of the represented system $\mathfrak{S}$, i.e., whose removal enlarges the set of valuations satisfying the formula representation $\varphi(\mathfrak{S})$ of $\mathfrak{S}$. For example, removing a clock constraint from a transition enriches the behaviour, in that the transition is enabled more often. Note that this is only true for convex clock constraints, because with logical conjunction $\wedge$ as the only logical connective (cf. Definition 2.1.2), removing any of the conjuncts removes a subconstraint, and thus enlarges the set of satisfying valuations.

The intended idea of the map of merging γ is to contain mappings for variables whose *mergence* in $\varphi(\mathfrak{S})$ enriches the behaviour of $\mathfrak{S}$. For example, merging two locations enriches the behaviour, in that the combined location allows for additional runs containing in- and outgoing transitions of different underlying locations.

We allow merging for propositional variables only, and omission for rational variables and arithmetic constraints. We first introduce some notation.

Notation 4.1.1 (Variable Sets (cf. Section 3.1.1)). For any real-time system $\mathfrak{S}$, let $\mathtt{S}$ and $\mathtt{X}$ be the sets of variables representing locations and clocks, respectively. For a TA $\mathfrak{A}$, let Σ be the set of variables representing events. For a TCA $\mathfrak{T}$, let $\mathtt{P_A}$, $\mathtt{P_{DA}}$, $\mathtt{D_F}$ and $\mathtt{D_{CO}}$ be the sets of variables representing port activity variables, port data variables, data fullness variables and data content variables, respectively. All variable sets are understood to be without indices.

We lift the notations from Section 2.1 in the straightforward way to reason about representation variables rather than constituents of real-time systems. In particular:

- by $\mathcal{CC}(\mathtt{X})$ and $\mathtt{X}|_{\mathtt{cc}}$, we denote the set of clock constraints over clock variables in $\mathtt{X}$, and the set of clock variables that occur in a clock constraint $\mathtt{cc}{\in}\mathcal{CC}(\mathtt{X})$, respectively (cf. Definition 2.1.2)

- by $\mathcal{DC}(\mathtt{P_{DA}},\mathtt{D_{CO}})$, $\mathtt{P_{DA}}|_{\mathtt{dc}}$ and $\mathtt{D_{CO}}|_{\mathtt{dc}}$, we denote the set of data constraints over port data variables in $\mathtt{P_{DA}}$ and data content variables in $\mathtt{D_{CO}}$, the set of port data variables that occur in a data constraint $\mathtt{dc}{\in}\mathcal{DC}(\mathtt{P_{DA}},\mathtt{D_{CO}})$, and the set of data content variables that occur in a data constraint $\mathtt{dc}{\in}\mathcal{DC}(\mathtt{P_{DA}},\mathtt{D_{CO}})$, respectively (cf. Definition 2.1.7)

- by $\mathcal{CC}(\mathtt{X})|_{\mathfrak{S}}$, we denote the set of clock constraints over clock variables in $\mathtt{X}$ that occur in the formula representation of a real-time system $\mathfrak{S}$; by $\mathcal{DC}(\mathtt{P_{DA}},\mathtt{D_{CO}})|_{\mathfrak{S}}$, we denote the set of data constraints over port data variables in $\mathtt{P_{DA}}$ and data content variables in $\mathtt{D_{CO}}$ that occur in the formula representation of $\mathfrak{S}$ (cf. Notations 2.2.3 and 2.3.4).

The exact nature of the map of merging γ and the set of omission $\mathcal{O}$ (i.e., to which system parts are γ and $\mathcal{O}$ applicable) depends on the the underlying system $\mathfrak{S}$. We now define these for TA and TCA separately.

Definition 4.1.2 (Map of Merging, Set of Omission for TA). Let $\mathfrak{A}$ be a TA, with formula representation $\varphi(\mathfrak{A})$, and variable sets as introduced in Notation 4.1.1, let $\mathtt{P}_{\mathfrak{A}}{=}(\mathtt{S}{\cup}\Sigma)$.

The *map of merging* $\gamma_{\mathfrak{A}}$ *of* $\mathfrak{A}$ is a total map $\gamma_{\mathfrak{A}}{:}\mathtt{P}_{\mathfrak{A}}{\to}(\mathtt{P}_{\mathfrak{A}}{\cup}\mathtt{P}'_{\mathfrak{A}})$, with $\mathtt{P}'_{\mathfrak{A}}$ some fresh set of propositional variables, and $\gamma_{\mathfrak{A}}(p){=}\gamma_{\mathfrak{A}}(p')$ only if $(p,p'{\in}\mathtt{S})$ or $(p,p'{\in}\Sigma)$. The *set of omission* $\mathcal{O}_{\mathfrak{A}}$ *of* $\mathfrak{A}$ is $\mathcal{O}_{\mathfrak{A}}{\subseteq}\mathtt{X}{\cup}\mathcal{CC}(\mathtt{X})$, where $\mathcal{CC}(\mathtt{X})$ only contains atomic formulas (i.e., which do not contain the logical operator $\wedge$, cf. Definition 2.1.2).

The map of merging $\gamma_{\mathfrak{A}}$ merges locations or actions, it is the identity for elements not intended to be merged. The additional constraint "only if $(p,p'{\in}\mathtt{S})$ or $(p,p'{\in}\Sigma)$" ensures that γ only maps variables to the same image if they are of the same conceptual type (a location cannot be not merged with an action). The set of omission $\mathcal{O}_{\mathfrak{A}}$ allows to remove clocks or single clock constraints. In the former case, the intended idea is to completely remove $\mathtt{x}$ from $\varphi(\mathfrak{A})$ (i.e., every occurrence of $\mathtt{x}$). For this reason, all clock constraints $\mathtt{cc}$ reasoning about $\mathtt{x}$, that means with $\mathtt{x}{\in}\mathtt{X}|_{\mathtt{cc}}$, need to be removed as well (even if $\mathtt{cc}$ itself is not contained in $\mathcal{O}_{\mathfrak{A}}$). The abstraction function (Definition 4.1.5) automatically takes care of this.

Definition 4.1.3 (Map of Merging, Set of Omission for TCA). Let $\mathfrak{T}$ be a TCA, with formula representation $\varphi(\mathfrak{T})$, and variable sets as introduced in Notation 4.1.1, let $\mathtt{P}_{\mathfrak{T}}{=}(\mathtt{S}{\cup}\mathtt{P}_{\mathtt{A}}{\cup}\mathtt{D}_{\mathtt{F}})$.

The *map of merging* $\gamma_{\mathfrak{T}}$ *of* $\mathfrak{T}$ is a total map $\gamma_{\mathfrak{T}}{:}\mathtt{P}_{\mathfrak{T}}{\to}(\mathtt{P}_{\mathfrak{T}}{\cup}\mathtt{P}'_{\mathfrak{T}})$, with $\mathtt{P}'_{\mathfrak{T}}$ some fresh set of propositional variables, and $\gamma_{\mathfrak{T}}(p){=}\gamma_{\mathfrak{T}}(p')$ only if $(p,p'{\in}\mathtt{S})$, or $(p,p'{\in}\mathtt{P}_{\mathtt{A}})$, or $(p,p'{\in}\mathtt{D}_{\mathtt{F}})$. The *set of omission* $\mathcal{O}_{\mathfrak{T}}$ *of* $\mathfrak{T}$ is $\mathcal{O}_{\mathfrak{T}}{\subseteq}\mathtt{X}{\cup}\mathcal{CC}(\mathtt{X}){\cup}\mathcal{DC}(\mathtt{P}_{\mathtt{DA}},\mathtt{D}_{\mathtt{CO}})$, where $\mathcal{CC}(\mathtt{X})$ and $\mathcal{DC}(\mathtt{P}_{\mathtt{DA}},\mathtt{D}_{\mathtt{CO}})$ do not contain compound formulas (i.e., do not contain logical operators $\wedge$ and $\neg$, cf. Definitions 2.1.2 and 2.1.7). In addition, if for some data constraint $\mathtt{dc}{\in}\mathcal{DC}(\mathtt{P}_{\mathtt{DA}},\mathtt{D}_{\mathtt{CO}})|_{\mathfrak{T}}$, there exists $\mathtt{Dp}{\in}\mathtt{P}_{\mathtt{DA}}|_{\mathtt{dc}}$ with $\gamma(\mathtt{p}){\neq}id$, or $\mathtt{Dd}{\in}\mathtt{D}_{\mathtt{CO}}|_{\mathtt{dc}}$ with $\gamma(\mathtt{d}){\neq}id$ (where $\mathtt{p}$ and $\mathtt{Dp}$ are port activity and port data variable of the same port p, and $\mathtt{d}$ and $\mathtt{Dd}$ are data fullness and data content variable of the same data variable d, cf. Section 3.1.1.4), we require $\mathtt{dc}{\in}\mathcal{O}_{\mathfrak{T}}$.

The map of merging $\gamma_{\mathfrak{T}}$ allows to merge locations, ports (actually port activity variables) or memory cells (actually data fullness variables), again it is the identity for elements not intended to be merged, and can only merge variables of the same conceptual type. The set of omission $\mathcal{O}_{\mathfrak{T}}$ allows to remove clocks and single clock constraints as before, and in addition allows to remove single data constraints. As for TA, the abstraction function (Definition 4.1.5) automatically takes care of removing all clock constraints $\mathtt{cc}$ with $\mathtt{x}{\in}\mathtt{X}|_{\mathtt{cc}}$ for a clock $\mathtt{x}{\in}\mathcal{O}_{\mathfrak{T}}$, even if $\mathtt{cc}{\notin}\mathcal{O}_{\mathfrak{T}}$.

For ports intended to be merged, i.e., with $\gamma(\mathtt{p}){\neq}id$, the automatic removal of data constraints by the abstraction function does not work in the same way as it does for clocks. The reason is that for a clock $\mathtt{x}$ and a clock constraint $\mathtt{cc}$, a simple syntactic check $\mathtt{x}{\in}\mathtt{X}|_{\mathtt{cc}}$ is sufficient to determine whether $\mathtt{cc}$ needs to be removed or not (see above). For ports, on the other hand, the set $\mathtt{P}_{\mathfrak{T}}$ contains port activity variables, while data constraints contain port data variables. Therefore, we need to explicitly add all data constraints to $\mathcal{O}_{\mathfrak{T}}$ that reason about ports intended to be merged. This is done by the last condition in Definition 4.1.3. The same argumentation holds for data fullness and data content variables.

Note that it is indeed necessary to *completely remove* data constraints that reason about ports and memory cells intended to be merged. The naive approach of replacing port data respectively data content variables in a data constraint by their image under $\gamma_{\mathfrak{T}}$ does not work, and may result in unsatisfiable data constraints. As an example, suppose a transition with port set $\{p, q\}$ and data constraint $((p{=}1){\wedge}(q{=}2))$, and suppose $\gamma_{\mathfrak{T}}(\mathtt{p}){=}\gamma_{\mathfrak{T}}(\mathtt{q}){=}\mathtt{r}$. Straightforward syntactic replacement of port data variables would yield a transition with port set $\{r\}$ and data constraint $((r{=}1){\wedge}(r{=}2))$. While the original (unabstracted) data constraint is satisfiable, the abstract data constraint is not; such abstraction would not yield an over-approximation.

Notation 4.1.4 (Abstraction). Since MO works uniformly on the different syntactical categories, in the sequel, we omit indices $\mathfrak{A}$ respectively $\mathfrak{T}$, and write γ, $\mathcal{O}$, $\mathtt{P}$ and $\mathtt{P}'$ only.

We use the term *domain of the abstraction*, denoted by ${}^\bullet\alpha$,[3] to refer to the set of parameters intended to be abstracted. That is, the domain of the abstraction consists of all elements in $\mathcal{O}$, and of those elements in $\mathtt{P}$ where γ is not the identity, that means which are mapped to an element in $\mathtt{P}'$. Formally, ${}^\bullet\alpha = \mathcal{O}{\cup}\gamma^{-1}(\mathtt{P}')$. Note that ${}^\bullet\alpha$ contains both (single) variables and (arithmetic) constraints.

We can now define the abstraction function.

Definition 4.1.5 (Abstraction by Merging Omission). Let $\mathfrak{S}$ be a real-time system, with $\varphi(\mathfrak{S})$ in NNF, and γ, $\mathcal{O}$, $\mathtt{P}$ and $\mathtt{P}'$ defined in Definitions 4.1.2 respectively 4.1.3.

The *abstraction of $\varphi(\mathfrak{S})$ (by merging omission) with respect to $\mathcal{O}$ and γ*, denoted as $\alpha_{\mathcal{O},\gamma}(\varphi(\mathfrak{S}))$, is defined in (4.4). We may write $\alpha(\varphi(\mathfrak{S}))$ if $\mathcal{O}$ and γ are clear form the context.

MO uniformly captures abstraction on all syntactic categories contained in $\varphi(\mathfrak{S})$: variables and constraints (cf. Section 3.2.1 for the definition of *Conts*$(\cdot)$) not meant to be abstracted (i.e., which are not contained in the domain of the abstraction ${}^\bullet\alpha$) are kept unchanged (4.1a). The map γ is applied to all positive propositional variables (4.1b). For negative propositional variables $\mathtt{p}$ (i.e., which occur as $\neg\mathtt{p}$ in $\varphi(\mathfrak{S})$) meant to be abstracted, we distinguish two cases: if there exists a positive propositional variable $\mathtt{p}'$ in the same conjunction as $\mathtt{p}$,[4] and with the same image under γ (i.e., $\mathtt{p}$ and $\mathtt{p}'$ are to be merged), we replace $\mathtt{p}$ with its positive image under γ (4.1c). The idea is that positive propositional variables are used to describe the "behaviour"—source and target of a transition, for example—while negative propositional variables are used to ensure consistency—mutual location exclusion,

[3]This notation anticipates Definition 4.1.5, where we use the symbol α to denote the abstraction function.

[4]*In the same conjunction* means enclosed by the same pair of parenthesis (note that some parenthesis can be omitted though, cf. Notation 2.1.1). For example, in $(\mathtt{p}{\wedge}\mathtt{p}'{\wedge}\mathtt{p}''){\vee}\mathtt{p}'''$, $\mathtt{p}$, $\mathtt{p}'$ and $\mathtt{p}''$ are in the same conjunction, but $\mathtt{p}'''$ is not.

$$\alpha'(L) = \begin{cases} L & \mathit{Conts}(L) \cap {}^\bullet\alpha = \emptyset \quad (4.1a) \\ \gamma(L) & \mathit{Conts}(L) \cap {}^\bullet\alpha \neq \emptyset, L = \mathtt{p} \in \mathtt{P} \quad (4.1b) \\ \gamma(L) & \mathit{Conts}(L) \cap {}^\bullet\alpha \neq \emptyset, L = \neg\mathtt{p}, \mathtt{p} \in \mathtt{P}, \quad (4.1c) \\ & \exists \mathtt{p}' \in \mathtt{P} \text{ in the same conjunction as } \mathtt{p} : \gamma(\mathtt{p}) = \gamma(\mathtt{p}') \\ \neg\gamma(L) & \mathit{Conts}(L) \cap {}^\bullet\alpha \neq \emptyset, L = \neg\mathtt{p}, \mathtt{p} \in \mathtt{P}, \quad (4.1d) \\ & \neg\exists \mathtt{p}' \in \mathtt{P} \text{ in the same conjunction as } \mathtt{p} : \gamma(\mathtt{p}) = \gamma(\mathtt{p}'), \\ & \exists \neg\mathtt{p}'', \mathtt{p}'' \in \mathtt{P}, \text{ in the same conjunction as } \mathtt{p} : \gamma(\mathtt{p}) = \gamma(\mathtt{p}'') \\ \mathtt{true} & \text{otherwise} \quad (4.1e) \end{cases}$$

$$\alpha'(F \wedge G) = \alpha'(F) \wedge \alpha'(G) \quad (4.2a)$$

$$\alpha'(F \vee G) = \alpha'(F) \vee \alpha'(G) \quad (4.2b)$$

$$\gamma_\alpha(\mathtt{P}) = \bigwedge_{p \in \gamma(\mathtt{P}) \setminus \mathtt{P}} \Big(\big((\bigwedge_{p' \in \gamma^{-1}(p)} \neg p') \vee p \big) \wedge \big(\bigvee_{p' \in \gamma^{-1}(p)} p' \vee \neg p \big) \Big) \quad (4.3)$$

$$\alpha(\varphi(\mathfrak{S})) = \alpha'(\varphi(\mathfrak{S})) \wedge \gamma_\alpha \quad (4.4)$$

Here, F and G are formulas in NNF, and L is a literal.

Figure 4.1: Abstraction by merging omission

for example. Therefore, if such $\mathtt{p}'$ exists, we can dismiss the literal $\neg\mathtt{p}$, since $\mathtt{p}$ and $\mathtt{p}'$ are mapped to the same image under γ, and we do not need the consistency constraint (for consistency between $\mathtt{p}$ and $\mathtt{p}'$) anymore. Note that replacing $\neg\mathtt{p}$ by $\mathtt{true}$ is possible as well, but this would yield a much coarser abstraction. If no such $\mathtt{p}'$ exists, but instead there exists a negative propositional variable $\mathtt{p}''$ with the same image under γ (4.1d), then we replace $\mathtt{p}$ and $\mathtt{p}''$ by their negative image under γ. Again, replacing $\neg\mathtt{p}$ and $\neg\mathtt{p}'$ by $\mathtt{true}$ is possible but would yield a much coarser abstraction. In all other cases, α' maps the literal to $\mathtt{true}$ (4.1e). In particular, this last case handles arithmetic constraints (both positive and negative) meant to be abstracted. Remember that arithmetic constraints not meant to be abstracted are handled in (4.1a) already.

In this way, α' performs a quick variant of existential abstraction [CGJ+03], while exploiting the syntactic categories and structural relationships of elemens in our formula representation.

In order to guarantee that MO yields an over-approximation, we need to keep track of the relation between symbols in $\mathtt{P}$ and their abstract counterparts in $\mathtt{P}'$. For this reason, we add the constraint γ_α (4.3) to the abstract formula (note that (4.3) is in NNF, and is equivalent to $\bigwedge_{p \in \gamma(\mathtt{P}) \setminus \mathtt{P}} \big((\bigvee_{p' \in \gamma^{-1}(p)} p') \leftrightarrow p \big)$).

As mentioned above after Definitions 4.1.2 and 4.1.3, the abstraction function automatically removes clock constraints $\mathtt{cc}$ over clocks $\mathtt{x} \in {}^\bullet\alpha$ (i.e., with $\mathtt{x} \in \mathtt{X}|_{\mathtt{cc}}$), even if $\mathtt{cc} \notin {}^\bullet\alpha$. The reason is that the only applicable rule in (4.1) for literal $\mathtt{cc}$ is (4.1e). Rule (4.1a) is not applicable, since the intersection $\mathit{Conts}(\mathtt{cc}) \cap {}^\bullet\alpha$ is not empty, and all other rules only handle propositional variables. Therefore, constraint $\mathtt{cc}$ is replaced by `true` by (4.1e).

In contrast, for ports intended to be abstracted (and equivalently for memory cells), the situation is different: a data constraint `dc` over port p contains the port data variable $\mathtt{Dp}{\in}\mathtt{P}_{\mathtt{DA}}$ (cf. Section 3.1.1), while the domain of the abstraction ${}^{\bullet}\alpha$ contains the port activity variable $\mathtt{p}{\in}\mathtt{P}_{\mathtt{A}}$. As a consequence, the intersection $\mathit{Conts}(\mathtt{dc}){\cap}{}^{\bullet}\alpha$ in (4.1a) would be empty. To correctly abstract from the data constraint, we explicitly add `dc` to ${}^{\bullet}\alpha$ (cf. Definition 4.1.3), such that the intersection is not empty anymore.

Notation 4.1.6 (Abstraction). Without confusion, in the sequel we use the symbol α only, and omit the symbol α'. For example, for a literal L, we write $\alpha(L)$ instead of $\alpha'(L)$.

We get the following results.

Lemma 4.1.7 (Abstraction by Weakening). Abstraction by merging omission, as defined in Definition 4.1.5, yields an over-approximation, that means $\alpha(F)$ is weaker than F in the sense that the implication $F{\rightarrow}\alpha(F)$ is *valid* (true in all models).

Proof. The proof can be found in Section A.2 in the Appendix, on Page 138. □

Theorem 4.1.8 (Correctness of Abstraction). Abstraction by merging omission, as defined in Definition 4.1.5, yields a correct over-approximation on sets of runs.

Proof. The proof can be found in Section A.2 in the Appendix, on Page 146. □

So far, we have assumed the variables to be without indices. Lifting α to the presence of localisations is straightforward: γ and $\mathcal{O}$ are understood oblivious to indices in the NNF of $\varphi(\mathfrak{S})$, such that indices directly carry over to $\varphi(\mathfrak{S})_k$ unchanged. Defining different abstractions for different steps is possible using the same definition of α, but we consider it to be less useful. Note that α is homomorphic with respect to $\{\wedge, \vee\}$, which proves the equality of $\alpha(\varphi(\mathfrak{S})_k)$ and $\alpha(\varphi(\mathfrak{S}))_k$ (except for speed of computing the abstraction, where $\alpha(\varphi(\mathfrak{S}))_k$ is superior).

Example 4.1.9 (Abstraction). Consider again the formula representation $\varphi(\mathfrak{A})$ of the intelligent light switch, as shown in Table 3.3 in Example 3.1.2. According to Definition 4.1.2, we have $\mathtt{P}{=}(\mathtt{S}{\cup}\Sigma){=}\{\mathtt{off}, \mathtt{light}, \mathtt{bright}, \mathtt{press}, \tau\}$. For merging locations `light` and `bright` into one location `on`, we define $\mathcal{O}{=}\emptyset$, $\mathtt{P}'{=}\{\mathit{on}\}$, $\gamma(\mathtt{light}){=}\gamma(\mathtt{bright}){=}\mathtt{on}$, and $\gamma(\mathtt{p}){=}\mathit{id}$ for $\mathtt{p}{\in}\mathtt{P}\backslash\{\mathtt{light}, \mathtt{bright}\}$.

The abstraction by merging omission of $\varphi(\mathfrak{A})$ with respect to γ and $\mathcal{O}$, $\alpha(\varphi(\mathfrak{A}))$, is shown in Table 4.2.

Note that we have simplified the formulas, by removing redundant parts. For example, after applying the abstraction function α, the first entry in Table 4.2

$$\begin{aligned}
\alpha(\varphi^{init}(\mathfrak{A})) &= \mathtt{off}_0\wedge\neg\mathtt{on}_0\wedge\neg\mathtt{press}_0\wedge\neg\mathtt{\tau}_0\wedge(\mathtt{z}_0{=}0)\wedge(\mathtt{x}_0{=}0)\\
\alpha(\varphi^{action}(e_1)) &= \mathtt{off}_\mathtt{t}\wedge\mathtt{press}_\mathtt{t+1}\wedge(\mathtt{z}_\mathtt{t}{=}\mathtt{z}_\mathtt{t+1})\wedge(\mathtt{x}_\mathtt{t+1}{=}\mathtt{z}_\mathtt{t+1})\wedge\mathtt{on}_\mathtt{t+1}\\
\alpha(\varphi^{action}(e_2)) &= \mathtt{on}_\mathtt{t}\wedge\mathtt{press}_\mathtt{t+1}\wedge(\mathtt{z}_\mathtt{t}{=}\mathtt{z}_\mathtt{t+1})\wedge(\mathtt{z}_\mathtt{t}-\mathtt{x}_\mathtt{t}{>}3)\wedge(\mathtt{x}_\mathtt{t+1}{=}\mathtt{x}_\mathtt{t})\wedge\mathtt{off}_\mathtt{t+1}\\
\alpha(\varphi^{action}(e_3)) &= \mathtt{on}_\mathtt{t}\wedge\mathtt{press}_\mathtt{t+1}\wedge(\mathtt{z}_\mathtt{t}{=}\mathtt{z}_\mathtt{t+1})\wedge(\mathtt{z}_\mathtt{t}-\mathtt{x}_\mathtt{t}{\leq}3)\wedge(\mathtt{x}_\mathtt{t+1}{=}\mathtt{x}_\mathtt{t})\wedge\mathtt{on}_\mathtt{t+1}\\
\alpha(\varphi^{action}(e_4)) &= \mathtt{on}_\mathtt{t}\wedge\mathtt{press}_\mathtt{t+1}\wedge(\mathtt{z}_\mathtt{t}{=}\mathtt{z}_\mathtt{t+1})\wedge(\mathtt{x}_\mathtt{t+1}{=}\mathtt{x}_\mathtt{t})\wedge\mathtt{off}_\mathtt{t+1}\\
\alpha(\varphi^{delay}(\mathit{off})) &= \mathtt{off}_\mathtt{t}\wedge\neg\mathtt{press}_\mathtt{t+1}\wedge\neg\mathtt{\tau}_\mathtt{t+1}\wedge(\mathtt{z}_\mathtt{t}{\leq}\mathtt{z}_\mathtt{t+1})\wedge(\mathtt{x}_\mathtt{t}{=}\mathtt{x}_\mathtt{t+1})\wedge\mathtt{off}_\mathtt{t+1}\\
\alpha(\varphi^{delay}(\mathit{light})) &= \mathtt{on}_\mathtt{t}\wedge\neg\mathtt{press}_\mathtt{t+1}\wedge\neg\mathtt{\tau}_\mathtt{t+1}\wedge(\mathtt{z}_\mathtt{t}{\leq}\mathtt{z}_\mathtt{t+1})\wedge(\mathtt{x}_\mathtt{t}{=}\mathtt{x}_\mathtt{t+1})\wedge\mathtt{on}_\mathtt{t+1}\\
&= \alpha(\varphi^{delay}(\mathit{bright}))\\
\alpha(\varphi^{trans}(\mathfrak{A})) &= \alpha(\varphi^{action}(e_1))\vee\alpha(\varphi^{action}(e_2))\vee\alpha(\varphi^{action}(e_3))\vee\alpha(\varphi^{action}(e_4))\vee\\
&\quad\ \alpha(\varphi^{delay}(\mathit{off}))\vee\alpha(\varphi^{delay}(\mathit{light}))\\
\alpha(\varphi^{location}(\mathfrak{A})) &= (\mathtt{off}_\mathtt{t+1}\wedge\neg\mathtt{on}_\mathtt{t+1})\vee(\mathtt{on}_\mathtt{t+1}\wedge\neg\mathtt{off}_\mathtt{t+1})\\
\alpha(\varphi^{mutex}(\mathfrak{A})) &= (\mathtt{press}_\mathtt{t+1}\wedge\neg\mathtt{\tau}_\mathtt{t+1})\vee(\mathtt{\tau}_\mathtt{t+1}\wedge\neg\mathtt{press}_\mathtt{t+1})\vee(\neg\mathtt{press}_\mathtt{t+1}\wedge\neg\mathtt{\tau}_\mathtt{t+1})\\
\gamma_\alpha(\mathtt{P}) &= ((\neg\mathtt{light}_\mathtt{t}\wedge\neg\mathtt{bright}_\mathtt{t})\vee\mathtt{on}_\mathtt{t})\wedge(\mathtt{light}_\mathtt{t}\vee\mathtt{bright}_\mathtt{t}\vee\neg\mathtt{on}_\mathtt{t})\\
\alpha(\varphi(\mathfrak{A})) &= \alpha(\varphi^{init}(\mathfrak{A}))\wedge\alpha(\varphi^{trans})\wedge\alpha(\varphi^{location}(\mathfrak{A}))\wedge\alpha(\varphi^{mutex}(\mathfrak{A}))\wedge\gamma_\alpha(\mathtt{P})
\end{aligned}$$

Table 4.2: Abstraction by Merging Omission: Example

contains conjunct $\neg\mathtt{on}_0$ twice, resulting from applying γ to $\neg\mathtt{bright}_0$ and $\neg\mathtt{light}_0$ in the original formula (cf. Table 3.3).

4.2 Concretisation

In the previous Section, we have defined abstraction by merging omission on the formula representation $\varphi(\mathfrak{S})$ of a real-time system $\mathfrak{S}$. We have explained how to obtain the abstract formula $\alpha(\varphi(\mathfrak{S}))$ from $\varphi(\mathfrak{S})$, by removing parameters that are considered irrelevant for the verification of a particular safety property.

The general problem with abstraction is that typically, it is not clear up front what is the set of relevant parameters that need to be kept in order to preserve correctness of verification results. It can therefore happen that one or more parameters from this set of relevant parameters are removed by the abstraction. If this happens, we get *false negatives*: a false negative is a "proof" that the property is violated in the abstract system, while in the original (concrete) system it is not (cf. the explanations at the beginning of this Chapter).

Thus, if a counterexample to the property under test has been found in the abstract system, we need to check whether this counterexample is real (i.e., corresponds to a true violation of the property, also in the original system) or spurious (i.e., is due to wrong abstraction, and the abstraction needs to be refined). To this end, the abstract counterexample is translated back into the original system, to determine whether it is reproducible there. This is called *concretisation.* In our context, concretisation works as follows.

First recall that to check whether a property expressed by formula ϕ holds in the abstract system $\alpha(\varphi(\mathfrak{S}))$, we check the conjunction $\alpha(\varphi(\mathfrak{S}))_k\wedge\neg\phi$ for satisfiability (cf. Section 3.2.3). If the formula is satisfiable, this indicates that the property does not hold in the abstract system, and every model σ of $\alpha(\varphi(\mathfrak{S}))_k\wedge\neg\phi$ corresponds to a run of the abstract system which violates the property.

Next, observe that every model σ of $\alpha(\varphi(\mathfrak{S}))_k\wedge\neg\phi$ is a total assignment to the variables in $\mathit{Vars}(\alpha(\varphi(\mathfrak{S}))_k\wedge\neg\phi)$, but only a partial assignment to the variables in $\mathit{Vars}(\varphi(\mathfrak{S})_k\wedge\neg\phi)$, that means some variables in $\varphi(\mathfrak{S})_k$ are left unconstrained by σ. This reflects the fact that one run in the abstract system can correspond to a set of runs in the original system.

Concretisation now consists in trying to extend the model σ of $\alpha(\varphi(\mathfrak{S}))_k\wedge\neg\phi$ to a model of $\varphi(\mathfrak{S})_k\wedge\neg\phi$, that agrees with σ on variables in $\mathit{Vars}(\alpha(\varphi(\mathfrak{S}))_k\wedge\neg\phi)$. This is done by interpreting σ as a conjunction of variable assignments:

$$\rho_\sigma = \bigwedge_{\substack{\mathtt{v}\in \mathit{Vars}(\alpha(\varphi(\mathfrak{S}))_k\wedge\neg\phi)\\ \sigma(\mathtt{v})=v}} (\mathtt{v}{=}v), \tag{4.5}$$

and trying to find a model for the conjunction $\varphi(\mathfrak{S})_k\wedge\neg\phi\wedge\rho_\sigma$. We call ρ_σ the *witness run (for ϕ)*. As the witness run is highly restrictive (it singles out only one abstract path), the abstract counterexample guides the search through the concrete system with a very narrow focus and is highly efficient. If such a model for $\varphi(\mathfrak{S})_k\wedge\neg\phi\wedge\rho_\sigma$ exists, that means, if the witness run is concretisable, we have found a run in the original system that violates the property. If no such model exists, i.e, $\varphi(\mathfrak{S})_k\wedge\neg\phi\wedge\rho_\sigma$ is unsatisfiable, the counterexample is spurious, and the abstraction needs to be refined.

Remark 4.2.1 (Concretisation). Since $\sigma\models(\alpha(\varphi(\mathfrak{S}))_k\wedge\neg\phi)$ (i.e., σ is a model of $\alpha(\varphi(\mathfrak{S}))_k\wedge\neg\phi$), in particular $\sigma\models\neg\phi$. By construction of ρ_σ, we have $\models\rho_\sigma\rightarrow\neg\phi$. For every valuation σ', we thus have $\sigma'\models\varphi(\mathfrak{S})_k\wedge\neg\phi\wedge\rho_\sigma$ iff $\sigma'\models\varphi(\mathfrak{S})_k\wedge\rho_\sigma$. For this reason, we can safely remove the conjunct $\neg\phi$ from the formula $\varphi(\mathfrak{S})_k\wedge\neg\phi\wedge\rho_\sigma$ during concretisation, and check the satisfiability of $\varphi(\mathfrak{S})_k\wedge\rho_\sigma$ only, without affecting the results. We will use this fact for interpolation.

In the next Section, we describe how to use Craig interpolants to derive information about which parameters need to be refined.

4.3 Interpolation

In this section, we introduce Craig interpolants (Section 4.3.1), and discuss their expressive power in the context of SAT-based verification (Section 4.3.2). We do *not* show how to derive/compute interpolants, since powerful tools exist for this task. The interested reader is referred to [McM03, McM05b], for example.

4.3.1 Craig Interpolants

Craig interpolants have been defined in [Cra57]. They are defined for pairs of inconsistent formulas (i.e., formulas which are unsatisfiable together), and provide a way of capturing the reason of inconsistency.

Definition 4.3.1 (Craig Interpolant). Let (F_1, F_2) be a pair of inconsistent formulas, i.e., with $\models \neg(F_1 \wedge F_2)$. A *Craig interpolant for* F_1 *and* F_2 (or simply *interpolant*) is a formula G such that

$$\models F_1 \rightarrow G, \tag{4.6}$$

$$\models G \rightarrow \neg F_2,^{5} \text{ and} \tag{4.7}$$

$$\mathit{Vars}(G) \subseteq \mathit{Vars}(F_1) \cap \mathit{Vars}(F_2). \tag{4.8}$$

Here, $\mathit{Vars}(\varphi)$ denotes the set of variables in a formula φ. F_1 is called *prefix of* G, and F_2 is called *suffix of* G.

For a pair of inconsistent formulas, an interpolant always exists [Cra57], and can be derived from a resolution proof of inconsistency of F_1 and F_2 in time linear in the size of the proof [Pud97, McM03, McM05b]. Originally [Cra57], interpolants were defined for purely propositional formulas, but it has been shown that they can be derived in the same way for formulas with linear (in)equalities over rational numbers, see [McM04] for details. Interpolants are not unique: for a pair of inconsistent formulas, typically many formulas exist which fulfil the conditions in Definition 4.3.1. Furthermore, resolution proofs are not unique either, that means the same pair of formulas can have different resolution proofs of inconsistency, depending on for example to which subformulas the resolution rule is applied first.

An interpolant G for an inconsistent pair of formulas (F_1, F_2) captures the reason of inconsistency as follows: G is always an over-approximation of the prefix F_1 (4.6), that means every model of F_1 is also a model of G. In this way, G captures the facts that can be derived from the prefix, that means the maximal set of facts that holds for every valuation of the prefix. Moreover, G is also an under-approximation of the negated suffix $\neg F_2$ (4.7), that means every model of G is also a model of $\neg F_2$. In this way, G captures facts that are inconsistent with F_2, that means the minimal set of facts that can never be extended to a satisfying valuation of the suffix. Since the interpolant contains only common symbols of prefix and suffix (4.8), it captures the cause of inconsistency, in that the constraints expressed by the interpolant are sufficient to show the inconsistency of prefix and suffix.

Interpolants are defined for pairs of formulas. If we have a single unsatisfiable formula F that is a conjunction of two subformulas, i.e., $F{=}F_1 \wedge F_2$, we can split F around this top-level conjunction, and derive an interpolant for the resulting pair (F_1, F_2). If F is a conjunction of more than two subformulas, i.e., $F{=}F_1 \wedge \ldots \wedge F_n$, for $n{>}2$, we can interpret F as a sequence of formulas $F_1, \ldots, F_n$, and derive an interpolant for every possible nonempty bisection (i.e., both subsequences contain at least one formula each) of the sequence. This corresponds to deriving an interpolant for every possible split around a top-level conjunction of F. In this way, we get a sequence of $n{-}1$ interpolants $G_1, \ldots, G_{n-1}$, such that G_i is an interpolant for the pair $(F_1 \wedge \ldots \wedge F_i, F_{i+1} \wedge \ldots \wedge F_n)$, $i \in \{1, \ldots, n{-}1\}$. In [McM05a], the author showed

[5]The notion $\models \neg(G \wedge F_2)$ is also commonly found in the literature, but in this context, we consider our (equivalent) notion $\models G \rightarrow \neg F_2$ more comprehensible.

that if the sequence of interpolants $G_1, \dots, G_{n-1}$ is derived from *the same* refutation proof for the inconsistency of $F_1, \dots, F_n$, then

$$\models (G_i \wedge F_{i+1}) \rightarrow G_{i+1} \tag{4.9}$$

holds for every $i \in \{1, \dots, n-1\}$.

There exist a number of tools that support interpolant generation for SMT formulas, like for example CSIsat [BZM08, csi], FOCI [FOC] or MathSAT [mat]. The former only supports interpolant generation for a pair of formulas. The latter two support the approach just described: when called on a sequence of n inconsistent formulas, they generate a sequence of $n-1$ interpolants with the property (4.9).

4.3.2 Expressiveness of Interpolants

In this section, we discuss what kind of information about the cause of unsatisfiability can be derived from interpolants, and we show how the sequential order of formulas can influence the generated interpolants.

For a pair of inconsistent formulas (F_1, F_2), we can derive a single interpolant G. Without further taking into account the concrete structure of prefix F_1 and suffix F_2, we can derive the following information about the inconsistency of F_1 and F_2 from G:

- G=`true`: we do not get any information about the prefix, since $F_1 \rightarrow$ `true` (4.6) holds for every F_1. We know that the suffix by itself is inconsistent, since `true` $\rightarrow \neg F_2$ (4.7) only evaluates to `true` if F_2 evaluates to `false`
- G=`false`: we know that the prefix by itself is inconsistent, since $F_1 \rightarrow$ `false` (4.6) only evaluates to `true` if F_1 evaluates to `false`. We do not get any information about the suffix, since `false` $\rightarrow F_2$ (4.7) holds for every F_2.
- G=F for some formula F, $F \neq$`true`, $F \neq$`false`: we do not get any information about prefix or suffix alone, but we know that the conjunction is unsatisfiable. As explained above, the constraints expressed by F are sufficient to prove the inconsistency of F_1 and F_2.

Obviously, changing the sequential order of F_1 and F_2— that means deriving an interpolant G' for the pair (F_2, F_1)—does not yield additional information about the cause of unsatisfiability. In particular, observe that if G is an interpolant for (F_1, F_2), then it follows directly from Definition 4.3.1 that $\neg G$ is an interpolant for (F_2, F_1). However, when deriving a sequence of $n-1$ interpolants from a sequence of n formulas, the sequential order of the formulas has a considerable influence on the generated interpolants, and thus on their expressiveness, as the following example shows.

Example 4.3.2 (Sequential Formula Order and Expressiveness of Interpolants). Consider two sequential orders for a set of inconsistent formulas:

$$(a \leftrightarrow b), (a \leftrightarrow c), (b \leftrightarrow e), (e \leftrightarrow f), (c \leftrightarrow d), (a \leftrightarrow \neg b), \text{ and} \tag{4.10}$$

$$(c \leftrightarrow d), (a \leftrightarrow c), (a \leftrightarrow b), (a \leftrightarrow \neg b), (b \leftrightarrow e), (e \leftrightarrow f). \tag{4.11}$$

The inconsistency is solely caused by the two formulas $(a\leftrightarrow b)$ and $(a\leftrightarrow\neg b)$. If we apply the SMT solver MathSAT to each of the two sequences, we get the interpolant sequences shown in Table 4.3 (for (4.10) on the left, for (4.11) on the right).[6]

formulas	interpolants	formulas	interpolants
$(a\leftrightarrow b)$		$(c\leftrightarrow d)$	
	$(a\leftrightarrow b)$		`true`
$(a\leftrightarrow c)$		$(a\leftrightarrow c)$	
	$(a\leftrightarrow b)\wedge(a\leftrightarrow c)$		`true`
$(b\leftrightarrow e)$		$(a\leftrightarrow b)$	
	$(a\leftrightarrow b)\wedge(a\leftrightarrow c)\wedge(b\leftrightarrow e)$		$(a\leftrightarrow b)$
$(e\leftrightarrow f)$		$(a\leftrightarrow\neg b)$	
	$(a\leftrightarrow b)$		`false`
$(c\leftrightarrow d)$		$(b\leftrightarrow e)$	
	$(a\leftrightarrow b)$		`false`
$(a\leftrightarrow\neg b)$		$(e\leftrightarrow f)$	

Table 4.3: Interpolant sequences for different formula orders

Notice that the interpolants derived for sequence (4.11) (on the right side of Table 4.3) are much simpler than the interpolants derived for (4.10). In particular, there is only one interpolant $\notin\{\texttt{true},\texttt{false}\}$ on the right side. This is due to the fact that the formulas in the sequence (4.11) are arranged in such a way that the number of common variables of prefix and suffix is minimised. As a result, the cause of unsatisfiability becomes clear immediately from the interpolants derived for (4.11): the third interpolant $(a\leftrightarrow b)$ (the only interpolant $\notin\{\texttt{true},\texttt{false}\}$) precisely describes the constraints that cause the inconsistency.

The interpolants derived for sequence (4.10) (on the left side of Table 4.3), on the other hand, are more complex,[7] because the number of common variables of prefix and suffix is larger. As a result, the cause of unsatisfiability is not immediately clear (though the repeated occurrence of interpolant $(a\leftrightarrow b)$ gives an indication already, but this is in general not the case for larger formula sequences and larger numbers of common variables).

The example has shown that reducing the number of common variables of prefix and suffix helps to obtain more expressive interpolants: in the best case, a number of interpolants simplify to `true` or `false`, which allows to reduce the sequence of formulas to an inconsistent subsequence. Moreover, less common variables of prefix and suffix—and thus less variables that can be contained in the interpolant—yield less complex invariants, which in turn capture the reason of inconsistency more precisely. To illustrate this last statement, consider the third interpolant $(a\leftrightarrow b)\wedge(a\leftrightarrow c)\wedge(b\leftrightarrow e)$

[6]Read the Table as follows: for each interpolant, the prefix of the interpolant consists of all formulas above it, and the suffix consists of all formulas below it. For example, for the second interpolant on the left, $(a\leftrightarrow b)\wedge(a\leftrightarrow c)$, the prefix is $(a\leftrightarrow b)\wedge(a\leftrightarrow c)$, and the suffix is $(b\leftrightarrow e)\wedge(e\leftrightarrow f)\wedge(c\leftrightarrow d)\wedge(a\leftrightarrow\neg b)$.

[7]More complex in the sense that they involve more variables, and have more satisfying interpretations.

derived for (4.10) (left side of Table 4.3), and the third interpolant $(a{\leftrightarrow}b)$ derived for (4.11) (right side of Table 4.3). While the latter precisely describes the constraints causing the inconsistency of (4.11) (as explained above), the former contains the superfluous conjuncts $(a{\leftrightarrow}c)$ and $(b{\leftrightarrow}e)$.

Note that the argumentation for interpolants `true` and `false` about the unsatisfiability of prefix and suffix directly carries over from pairs of formulas. That is, the suffix of an interpolant `true` (which is now a set of formulas) is inconsistent by itself, and so is the prefix of an interpolant `false`. For example, for the second interpolant `true` on the right side of Table 4.3, the suffix $(a{\leftrightarrow}b){\wedge}(a{\leftrightarrow}\neg b){\wedge}(b{\leftrightarrow}e){\wedge}(e{\leftrightarrow}f)$ is inconsistent by itself. We will come back to this fact in the next section.

4.3.3 Sequential Formula Order for $\varphi(\mathfrak{S})$

We now present a sequential formula order for the subformulas of $\varphi(\mathfrak{S})$ that takes into account the considerations from the previous Section.

Remember that we need to refine the abstraction if the witness run ρ_σ represents a spurious counterexample, that means if the conjunction $\alpha(\varphi(\mathfrak{S}))_k{\wedge}\neg\phi$ (of abstract system and property) is satisfiable, but the conjunction $\varphi(\mathfrak{S})_k{\wedge}\rho_\sigma$ (of original system and witness run) is not, cf. Section 4.2 and in particular Remark 4.2.1. To find the parameters that were wrongly abstracted, we derive a sequence of interpolants for $\varphi(\mathfrak{S})_k{\wedge}\rho_\sigma$, and from these determine the ill-abstracted parameters. To increase the expressiveness of the interpolants, we take into account the results from the previous section, by syntactically reordering the subformulas of $\varphi(\mathfrak{S})_k{\wedge}\rho_\sigma$ (without changing the semantics), such that the number of common variables is minimised. Intuitively, the resulting sequence results from "interleaving" elements from $\varphi(\mathfrak{S})_k$ and ρ_σ, based on their unfolding depth. In detail, we construct the sequence as follows.

First, we split $\varphi(\mathfrak{S})_k$ around top-level conjunctions, based on the partition in (3.29), and reconjunct elements where all variables have the same unfolding depth, resulting in the following set

$$\begin{aligned}\{\varphi^{init}(\mathfrak{S}), \varphi^{trans}(\mathfrak{S})_{(\mathtt{0})}, \varphi^{location}(\mathfrak{S})_{(\mathtt{1})}{\wedge}\varphi^{mutex}(\mathfrak{S})_{(\mathtt{1})}, \ldots\\ \ldots, \varphi^{trans}(\mathfrak{S})_{(\mathtt{k-1})}, \varphi^{location}(\mathfrak{S})_{(\mathtt{k})}{\wedge}\varphi^{mutex}(\mathfrak{S})_{(\mathtt{k})}\}\end{aligned} \tag{4.12}$$

Next, we do the same for the witness run: we split ρ_σ around the conjunctions (cf. (4.5)), and reconjunct elements (variable assignments) with the same unfolding depth, which yields the set

$$\{\bigwedge_{\substack{\mathtt{v_0}\in Vars(\rho_\sigma)\\ \sigma(\mathtt{v_0})=v}}(\mathtt{v_0}{=}v), \ldots, \bigwedge_{\substack{\mathtt{v_k}\in Vars(\rho_\sigma)\\ \sigma(\mathtt{v_k})=v}}(\mathtt{v_k}{=}v)\} \tag{4.13}$$

Finally, we join the two sets, conjunct elements with the same unfolding depth, and stratify (i.e., sort by unfolding depth) the formulas. The result is the following sequence of formulas

$$
\begin{aligned}
&\varphi^{init}(\mathfrak{S}) \wedge \bigwedge_{\substack{\mathtt{v}_0 \in \mathit{Vars}(\rho_\sigma) \\ \sigma(\mathtt{v}_0)=v}} (\mathtt{v}_0 = v), \\
&\varphi^{trans}(\mathfrak{S})_{(\mathtt{0})}, \\
&\varphi^{location}(\mathfrak{S})_{(\mathtt{1})} \wedge \varphi^{mutex}(\mathfrak{S})_{(\mathtt{1})} \wedge \bigwedge_{\substack{\mathtt{v}_1 \in \mathit{Vars}(\rho_\sigma) \\ \sigma(\mathtt{v}_1)=v}} (\mathtt{v}_1 = v), \\
&\varphi^{trans}(\mathfrak{S})_{(\mathtt{1})}, \ldots, \\
&\varphi^{trans}(\mathfrak{S})_{(\mathtt{k-1})}, \\
&\varphi^{location}(\mathfrak{S})_{(\mathtt{k})} \wedge \varphi^{mutex}(\mathfrak{S})_{(\mathtt{k})} \wedge \bigwedge_{\substack{\mathtt{v}_\mathtt{k} \in \mathit{Vars}(\rho_\sigma) \\ \sigma(\mathtt{v}_\mathtt{k})=v}} (\mathtt{v}_\mathtt{k} = v)
\end{aligned}
\tag{4.14}
$$

In the sequel, without confusion we may use $\varphi(\mathfrak{S})_k \wedge \rho_\sigma$ to refer to the "reordered" variant (4.14). For example, by "the sequence of interpolants derived for $\varphi(\mathfrak{S})_k \wedge \rho_\sigma$" (in Section 4.4, for example), we actually mean to the sequence of interpolants derived for (4.14).

Reordering the subformulas of $\varphi(\mathfrak{S})_k \wedge \rho_\sigma$ in this way has two major advantages. The first advantage is that we have minimised the number of common as much as possible: elements of the sequence alternately contain variables of one respectively two unfolding depths,[8] and due to the stratification, every two subsequent elements of the sequence share variables of exactly one unfolding depth. In this way, every interpolant derived for any bisection of the sequence into prefix and suffix can contain variables of (at most, cf. interpolants `true` and `false`) one unfolding depth. Minimising the number of common variables has the advantages illustrated in the previous Section.

The second advantage is that prefixes and suffixes of any interpolant directly correspond to prefixes and suffixes of the witness run: for any interpolant G_i, with $1 \le i \le 2k$,[9] a model of the prefix of G_i directly corresponds to a prefix of length $\lfloor \frac{i}{2} \rfloor$ of the run through $\mathfrak{S}$ represented by the witness run ρ_σ. Again, this is due to the fact that we stratified the formulas in (4.14). If $G_i = \mathtt{false}$, we can thus conclude that the prefix of length $\lfloor \frac{i}{2} \rfloor$ of the witness run, which is represented by those elements of (4.13) with unfolding depth smaller or equal to $\lfloor \frac{i-1}{2} \rfloor$, is not concretisable. Equivalently, if $G_i = \mathtt{true}$, we can conclude that the suffix of length $\lceil \frac{2k-i}{2} \rceil$ of the witness run, which is represented by those elements of (4.13) with unfolding depth greater or equal to $\lceil \frac{i}{2} \rceil$, is not concretisable.

Remark 4.3.3. To illustrate the bounds in the above explanations (for example, to illustrate that the prefix of interpolant G_i indeed corresponds to a prefix of the witness run of length $\lfloor \frac{i}{2} \rfloor$), consider the following example. For unfolding depth

[8]In (4.14), odd-numbered elements contain variables of one unfolding depth. For example, the first element $\varphi^{init}(\mathfrak{S}) \wedge \bigwedge_{\mathtt{v}_0 \in \mathit{Vars}(\rho_\sigma), \sigma(\mathtt{v}_0)=v} (\mathtt{v}_0 = v)$ contains variables of unfolding depth 0 only. Even-numbered elements contain variables of two unfolding depths. For example, the second element $\varphi^{trans}(\mathfrak{S})_{(\mathtt{0})}$ contains variables of unfolding depths 0 and 1.

[9]Observe that for unfolding depth `k`, the sequence (4.14) has $2k+1$ elements, which allows to derive a sequence $G_1, \ldots, G_{2k}$ of $2k$ interpolants.

3, the sequence (4.14) has 2∗3+1 = 7 elements, for which we can derive 2∗3 = 6 interpolants $G_1, \ldots, G_6$:

$$\begin{array}{ll}
\varphi^{init}(\mathfrak{S}) \wedge \bigwedge_{\mathtt{v}_0 \in Vars(\rho_\sigma), \sigma(\mathtt{v}_0)=v} (\mathtt{v}_0 = v) & \\
& G_1 \\
\varphi^{trans}(\mathfrak{S})_{(\mathtt{0})} & \\
& G_2 \\
\varphi^{location}(\mathfrak{S})_{(\mathtt{1})} \wedge \varphi^{mutex}(\mathfrak{S})_{(\mathtt{1})} \wedge \bigwedge_{\mathtt{v}_1 \in Vars(\rho_\sigma), \sigma(\mathtt{v}_1)=v} (\mathtt{v}_1 = v) & \\
& G_3 \\
\varphi^{trans}(\mathfrak{S})_{(\mathtt{1})} & \\
& G_4 \\
\varphi^{location}(\mathfrak{S})_{(\mathtt{2})} \wedge \varphi^{mutex}(\mathfrak{S})_{(\mathtt{2})} \wedge \bigwedge_{\mathtt{v}_2 \in Vars(\rho_\sigma), \sigma(\mathtt{v}_2)=v} (\mathtt{v}_2 = v) & \\
& G_5 \\
\varphi^{trans}(\mathfrak{S})_{(\mathtt{2})} & \\
& G_6 \\
\varphi^{location}(\mathfrak{S})_{(\mathtt{3})} \wedge \varphi^{mutex}(\mathfrak{S})_{(\mathtt{3})} \wedge \bigwedge_{\mathtt{v}_3 \in Vars(\rho_\sigma), \sigma(\mathtt{v}_3)=v} (\mathtt{v}_3 = v) &
\end{array}$$

Consider the case i=3: the prefix of interpolant G_3 consists of three formulas, which correspond to a run of length $\lfloor \frac{3}{2} \rfloor = 1$, and this run is represented by the variable valuations (i.e., the elements of (4.13)) with unfolding depths smaller or equal to $\lfloor \frac{3-1}{2} \rfloor = 1$. The suffix of G_3 consists of four formulas, which correspond to a run of length $\lceil \frac{6-3}{2} \rceil = 2$, and this run is represented by the variable valuations with unfolding depths greater or equal to $\lceil \frac{3}{2} \rceil = 2$.

4.4 Refinement

If a counterexample to the property under test has been detected in the abstract system, and this counterexample has turned out to be spurious in the concretisation step, the abstraction is too coarse and needs to be refined.

Most refinement approaches are based on information obtained from the spurious counterexample; the most well-known technique is called *counterexample-guided abstraction refinement (CEGAR)* [CGJ+03]. In CEGAR, one spurious abstract counterexample (corresponding to a set of runs in the concrete system, cf. Section 4.2) is ruled out within every refinement step, that means the abstraction is modified in such a way that the spurious counterexample is not feasible anymore.

We propose a variant of CEGAR, by defining two possibilities of refining the abstraction, both based on information obtained from the spurious counterexample.

4.4.1 Ruling Out a Counterexample Trace

The first refinement option is to rule out the spurious counterexample just found. The spurious counterexample is given by the set of valuations that constitute the witness run ρ_σ, cf. (4.5). To ensure that the behaviour expressed by the witness run is not feasible anymore in future verification steps, we could simply add $\neg\rho_\sigma$ as an additional conjunct to the representation of the abstract system $\alpha(\varphi(\mathfrak{S}))_k$, and, instead of checking $\alpha(\varphi(\mathfrak{S}))_k\wedge\neg\phi$ (cf. Section 4.2), model check $\alpha(\varphi(\mathfrak{S}))_k\wedge\neg\rho_\sigma\wedge\neg\phi$ in the next verification step. This is essentially equivalent to basic CEGAR.

Yet, we can do better, by taking into account information obtained from the interpolants. Let $G_1,\ldots,G_n$ be the sequence of interpolants derived for $\varphi(\mathfrak{S})_k\wedge\rho_\sigma$, let G_f, $1\leq f\leq n$, be the first interpolant in the sequence which is equal to `false`, and let G_t, $1\leq t\leq n$, be the last interpolant in the sequence which is equal to `true` (note that G_t and G_f do not necessarily exist).

If G_f exists, we know (cf. Definition 4.3.1 and Section 4.3.3) that the prefix of G_f is unconcretisable. Let i be the unfolding depth of variables contained in the interpolant G_{f-1}, which is the interpolant directly preceding G_f (remember that every interpolant contains variables of at most one unfolding depth). We reduce the witness run ρ_σ of length k to a subset $\rho_{\sigma_{\leq i}}$ of length i containing only variable valuations of variables with unfolding depth i or smaller, and model check $\alpha(\varphi(\mathfrak{S}))_k\wedge\neg\rho_{\sigma_{\leq i}}\wedge\neg\phi$ in the next verification step. In this way, in a single refinement step, we not only rule out *one* abstract counterexample, as is done in basic CEGAR, but we rule out *a set* of abstract counterexamples at once.

If G_t exists, we can use this interpolant in a similar way: let j be the unfolding depth of variables contained in the interpolant G_{t+1}, which is the interpolant directly following G_t. We reduce the witness run ρ_σ of length k to a subset $\rho_{\sigma_{\geq j+1}}$ of length $k-j+1$ containing only variable valuations of variables with unfolding depth $j+1$ or larger, and model check $\alpha(\varphi(\mathfrak{S}))_k\wedge\neg\rho_{\sigma_{\geq j+1}}\wedge\neg\phi$ in the next verification step.

If neither G_f nor G_t exists, or if $G_t=G_{f-1}$ (the latter typically only happens with small toy examples), this refinement step is not applicable. If both G_f and G_t exists, and $G_t\neq G_{f-1}$, we choose the interpolant that leads to a shorter counterexample fragment being ruled out, that is, we choose G_f if $i<k-j+1$ (with i,j as above), and G_t otherwise. The underlying idea is that a shorter witness run fragment corresponds to a larger set of witness runs of the full length k, and thus a larger set of counterexamples is ruled out at once.

4.4.2 Refining a Previously Abstracted Parameter

The second refinement option is to reduce the domain of the abstraction $^\bullet\alpha$, by removing a parameter from it (that means, either remove a parameter from $\mathcal{O}$, or remove a mapping from γ). In this way, the parameter is put back into the abstract system, such that it is considered again in future verification steps. More concretely, the current abstraction $\alpha_{\mathcal{O},\gamma}$ is transformed into a new abstraction $\widetilde{\alpha}_{\widetilde{\mathcal{O}},\widetilde{\gamma}}$ with $^\bullet\widetilde{\alpha}_{\widetilde{\mathcal{O}},\widetilde{\gamma}}\subset{}^\bullet\alpha_{\mathcal{O},\gamma}$, where either $\widetilde{\mathcal{O}}\subset\mathcal{O}$, or $\widetilde{\gamma}^{-1}(\mathtt{P}')\subset\gamma^{-1}(\mathtt{P}')$. The new abstraction $\widetilde{\alpha}_{\widetilde{\mathcal{O}},\widetilde{\gamma}}$ is computed as follows.

To refine a parameter $\mathtt{e}\in\mathcal{O}$ (a clock, a clock constraint or a data constraint), the parameter is simply removed from $\mathcal{O}$, that means $\widetilde{\mathcal{O}}=\mathcal{O}\backslash\mathtt{e}$. However, it is only possible to refine a clock constraint `cc` over a clock $\mathtt{x}\in\mathtt{X}|_{\mathtt{cc}}$ if `x` itself is not abstracted, that means $\mathtt{x}\notin{}^\bullet\alpha$, since otherwise, the new abstraction $\widetilde{\alpha}_{\widetilde{\mathcal{O}},\widetilde{\gamma}}$ would produce a system that contains a clock constraint over an undefined clock. Equivalently, it is only possible to refine a data constraint `dc` over over a port `p` with $\mathtt{Dp}\in\mathtt{P}_{\mathtt{DA}}|_{\mathtt{dc}}$, or over a memory cell `m` with $\mathtt{Dm}\in\mathtt{D}_{\mathtt{CO}}|_{\mathtt{dc}}$ if `p` and `m` themselves are not abstracted, that means $\mathtt{m}\notin{}^\bullet\alpha$, and $\mathtt{p}\notin{}^\bullet\alpha$.

If, instead, a parameter $\mathtt{e}{\in}\gamma^{-1}(\mathtt{P}')$ (a location, an event, a port or a memory cell) is to be refined, the mapping $\{\mathtt{e}{\mapsto}\mathtt{e}'\}$, with $\mathtt{e}'{\in}\mathtt{P}'$, is removed from γ and replaced by the identity mapping. If $\mathtt{e}$ was merged with one other parameter $\tilde{\mathtt{e}}$ only, that means there exists exactly one $\tilde{\mathtt{e}}{\in}\mathtt{P}$ with $\gamma(\tilde{\mathtt{e}}){=}\gamma(\mathtt{e}){=}\mathtt{e}'$, then the mapping for $\tilde{\mathtt{e}}$ is removed as well: $\widetilde{\gamma}{=}\gamma \oplus \{\mathtt{e}{\mapsto}\mathtt{e}, \tilde{\mathtt{e}}{\mapsto}\tilde{\mathtt{e}}\}$, where $\oplus$ denotes function overriding.[10] Otherwise, that means if $|\gamma^{-1}(\gamma(\mathtt{e}))| > 2$, no further changes need to be made to γ: $\widetilde{\gamma}{=}\gamma \oplus \{\mathtt{e}{\mapsto}\mathtt{e}\}$.

To solve the question which parameters from $^\bullet\alpha$ are to be considered for refinement, that means to identify the ill-abstracted parameters, we consider the set $\mathit{Conts}(G_f){\cap}^\bullet\alpha \overset{\text{def}}{=} {}^\bullet\alpha|_{G_{f-1}}$, with G_{f-1} as before, of elements which are potentially responsible for the spurious counterexample. Alternatively, we can consider the set $^\bullet\alpha|_{G_{t+1}}$, with G_{t+1} as before. After choosing one of the parameters from any of the sets, we refine it according to the procedure explained above. In the next verification step, the property is checked on the abstract system computed with the new abstraction function $\widetilde{\alpha}_{\widetilde{\mathcal{O}},\widetilde{\gamma}}$, that means model checking is applied to $\widetilde{\alpha}_{\widetilde{\mathcal{O}},\widetilde{\gamma}}(\varphi(\mathfrak{S}))_k{\wedge}\neg\phi$.

Note that we could actually take any interpolant for this refinement option, that means consider any set $^\bullet\alpha|_{G_i}$, with $1{\le}i{\le}n$, since by definition, every interpolant captures the reason of inconsistency of its prefix and suffix (cf. Definition 4.3.1 and Section 4.3.2). However, since $\mathit{Conts}(G_f) = \mathit{Conts}(G_t) = \emptyset$, choosing either G_f or G_t is not reasonable. Choosing any interpolant $G_{f'}$, with $f'{>}f$ (that means, which occurs in the sequence $G_1, \ldots, G_n$ at some point after G_f), is not reasonable either: the prefix of G_f is inconsistent by itself, that means the cause of unsatisfiability is entirely contained in this prefix. An interpolant $G_{f'}$, whose prefix contains the prefix of G_f, can therefore not provide more information about the cause of unsatisfiability. A similar argumentation holds for interpolants $G_{t'}$, with $t'{<}t$, which occur in the sequence $G_1, \ldots, G_n$ at some point before G_t. Note that in SMT solver tools like [csi, FOC, mat], such interpolants $G_{f'}$ and $G_{t'}$ are simplified to `false` respectively `true`, even if the solver derived an interpolant $\neq$`false` respectively $\neq$`true` for the particular bisection.

Let $F_1, \ldots, F_{n+1}$ be the sequence of formulas in $\alpha(\varphi(\mathfrak{S}))_k{\wedge}\neg\phi$ for which the interpolants $G_1, \ldots, G_n$ are derived. The reason why we choose G_{f-1} to determine candidates for refinement is the following: we know that the prefix of G_f (the set of formulas $F_1, \ldots, F_f$) is unsatisfiable, but the prefix of G_{f-1} (the set of formulas $F_1, \ldots, F_{f-1}$) is not. Thus, the addition of "suffix" F_f to the prefix $F_1, \ldots, F_{f-1}$ causes the whole sequence $F_1, \ldots, F_f$ to be unsatisfiable, and interpolant G_f describes the cause of this unsatisfiability (cf. also (4.9)). The argumentation for choosing G_{t+1} is similar.

4.4.3 Refinement Heuristics

For conciseness of explanation, we refer to the refinement option presented in Section 4.4.1 as the *first* (refinement) option, and to the refinement option presented in

[10]The mapping for $\tilde{\mathtt{e}}$ could actually remain in γ, since a single mapping $\{\tilde{\mathtt{e}}{\mapsto}\mathtt{e}'\}$, with $|\gamma^{-1}(\gamma(\tilde{\mathtt{e}}))| = 1$, corresponds to pure syntactic replacement of $\tilde{\mathtt{e}}$ by $\mathtt{e}'$, and therefore does not change the satisfiability of the result. Yet, $\tilde{\mathtt{e}}$ is a parameter in the original system $\varphi(\mathfrak{S})$, while $\mathtt{e}'$ is not, therefore, we prefer to remove $\mathtt{e}'$.

Section 4.4.2 as the *second* (refinement) option.

The problem with the second refinement option is *which* parameter to choose from $^\bullet\alpha|_{G_{f-1}}$ (respectively from $^\bullet\alpha|_{G_{t+1}}$) for refinement: since interpolants are not unique, typically not all parameters in the set are responsible for the present spurious counterexample, and some of the parameters might not be ill-abstracted at all. As an example, consider again the interpolants in the left column of Table 4.3: the second and the third interpolant contain parameters b and c, respectively b, c and e, which are completely irrelevant to the cause of inconsistency of the formulas in (4.10). Additionally, it is in general not clear under which conditions to apply either the first or the second refinement option. Thus, the remaining difficulty is to define heuristics describing the application of the two refinement options. Finding adequate heuristics is a problem common to almost all refinement approaches, cf. for example [CGJ+03, CCK+02].

While the first refinement option is quick and easy to apply, in that the application is straightforward once a counterexample has been found, it cannot yield results as long as essential parameters are inadequately abstracted. Application of the second option is not so straightforward and in addition involves more computational effort (recomputing the abstraction), but this option is indispensable to add ill-abstracted parameters back into the system. However, if the second option is applied too frequently, the abstract system quickly collapses to the original system. It is thus necessary to define heuristics that strike a suitable balance between options one and two.

The best solution to solving this problem is to leave the decision (which option and—for option two—which parameter to choose) to the human experts. The reasoning is that system developers who have designed and improved the system in a number of iterations in the development process have a deep insight into the functioning of the system, and, when presented with a counterexample and/or a set of potentially responsible parameters, these experts will be able to deduce information about the quality/applicability of the different refinement options.

To support the developers in reaching a decision, and as a first step towards automatic abstraction refinement, we propose the following heuristic, which in particular uses both refinement options one and two. We first rule a fixed number of counterexamples (for example $\frac{k}{2}$), using option one, and in every iteration record the set $^\bullet\alpha|_{G_{f-1}}$ (respectively $^\bullet\alpha|_{G_{t+1}}$) of parameters that are potentially responsible. Let $^\bullet\alpha|_{G^i_{f-1}}$ denote the set obtained in the i-th iteration. After this, we inspect the multiset $^\bullet\alpha|_{G^1_{f-1}} \cup {}^\bullet\alpha|_{G^2_{f-1}} \cup \ldots \cup {}^\bullet\alpha|_{G^n_{f-1}}$, and determine the parameter(s) with the highest multiplicity. If there exists a single such parameter, we refine it with refinement option two. Otherwise, if there are two or more parameters with the same (highest) multiplicity, we can either randomly choose a parameter among these, or continue applying option one until one parameter in the multiset has a higher multiplicity than the others.

The idea of this heuristic is as follows: as explained above, applying refinement option one is quick and easy, moreover, it will never result in an abstraction that is too fine (in contrast, when refining a parameter with option two that is irrelevant to the property, the abstraction becomes unnecessarily fine). We can therefore apply option one a number of times, without loosing efficiency. By taking into account the set of potentially responsible parameters ${}^{\bullet}\alpha|_{G_{f-1}}$ obtained from (many) different iterations, we increase the probability of choosing a parameter that is indeed ill-abstracted, since ill-abstracted parameters will occur more frequently.

4.5 Conclusion

In this Chapter, we have presented a general framework for abstraction and refinement of TA and TCA that are represented in propositional logic with linear arithmetic.

In Section 4.1, we have defined our uniform abstraction function. It is essentially equivalent to the abstraction function presented in [Kem11] (which in turn is an improved variant of the abstraction functions presented in [KP07, Kem09]). The major difference between the abstraction function presented here and the one in [KP07, Kem09] is the fact that we do *not* in general map negative propositional variables to `true`. Such an abstraction function would effectively remove the mutual exclusion constraint for locations (cf. (3.5), (3.14) and (3.24)) and TA events (3.6), as well as part of the consistency constraint on data values in TCA and TNA ((3.15) and (3.25)). Instead, we take into account the special characteristics of our formula representation, and the syntactic context of the propositional variable (4.1c), (4.1d), and in this way retain more information. See Section A.2 for further details.

In addition, while the abstraction functions presented in previous work were tailored to one system type each, we have shown in Section 4.1 that the abstraction function presented here uniformly handles both TA and TCA. We have provided correctness results, which in particular take into account the new representation features of memory cells and data values. In Section 4.2, we have restated in more detail the concept of concretisation, which has been briefly sketched in previous work [KP07, Kem11].

Section 4.3 deals with Craig interpolation. After a brief introduction to Craig interpolants, presented mainly for completeness, we have provided an extensive discussion on expressiveness of interpolants, and the type of information that can be derived from (a sequence of) interpolants. We use these results in Section 4.3.3 to reorder the formulas in the representation of TA and TCA such that the information derived from interpolants is maximised.

Finally, in Section 4.4, we have discussed two refinement options in detail. While these options were sketched in previous work already ([KP07, Kem11]), we here provide a detailed discussion of how and when to apply each of the options, and discuss advantages and disadvantages. Section 4.4.3 discusses refinement heuristics based on the two refinement options. Apart from automatic abstraction refinement, these heuristics can also be used to support user decisions on which refinement option to choose.