

Refinement for Predicate Abstraction in the Context of Abstract Component Model

Tonglaga Bao
 Department of Computer Science
 Brigham Young University
 Provo, UT 84602
 tonga@cs.byu.edu

Mike Jones
 Department of Computer Science
 Brigham Young University
 Provo, UT 84602
 jones@cs.byu.edu

This technical report presents the computational model and theoretical background of model checking as abstract component in the context of concrete software environment.

1. COMPUTATIONAL MODEL

In this section, we give a model of computation in which we can reason about the verification of abstract components in the context of concrete software.

Component A component is a reusable piece of code that can accomplish a certain task and which is designed to interact with other components or programs. In this work, we simply define a component as a range of program counter (pc) values. We let PC_c denote the set of pc values that belong to the component under test. The problem of extracting a component from a software artifact is important, but left as future work.

Given a complete software artifact and a PC_c range, our technique performs abstraction in the part of the program which corresponds to PC_c and leaves the rest of the software concrete. As such, our model of a system consists of a concrete part and an abstract part.

Concrete Model Given a finite set of atomic propositions AP, the state space of a program is modeled as a transition system $M = (S, s_0, T, L)$ where:

- S is a finite set of states,
- $s_0 \in S$ is the initial state,
- $T \subseteq S \times S$ is a set of transitions,
- $L : S \rightarrow 2^{AP}$ is a labeling function, where $L(s) = \{p \in AP \mid s \models p\}$

Abstract Model (adapted from [?]) Given a concrete program model $M = (S, s_0, T, L)$, a set of predicates $\Phi = \{\phi_1, \dots, \phi_n\}$, and an abstraction function $\alpha_\Phi : S \rightarrow B_n$ in which $B_n = \{b_1 \dots b_n\}$ is a set of bit vectors where $\alpha_\Phi(s) = b_1 \dots b_n$ with $b_i = 1$ if $s \models \phi_i$ else $b_i = 0$, the abstract model $A = (S_\alpha, \alpha_0, T_\alpha, L_\alpha)$ where:

- $S_\alpha = \{\alpha \mid s \in S \text{ and } \alpha = \alpha_\Phi(s)\}$ is a set of abstract states,
- $\alpha_0 = \alpha_\Phi(s_0)$ is the initial abstract state.
- $T_\alpha \subseteq S_\alpha \times S_\alpha$ is a set of abstract transitions.
- $L_\alpha : S_\alpha \rightarrow 2^{AP}$ is a labeling function for abstract states, where $L_\alpha(\alpha) = \{p \in AP \mid \alpha \models p\}$

Mixed Model The mixed model includes both the abstract and concrete parts of the state space. The definition of the mixed model requires the ability to extract the pc value from a system state. We define a function $V_{pc} : S \rightarrow Z$ such that $V_{pc}(s) = v$ where v is the pc value in state s . Given a concrete program model $M = (S, s_0, T, L)$, a predicate set Φ , and an abstraction function α_Φ , the mixed model $X = \{S_x, x_0, T_x, L_x\}$ is

- $S_x = \{x = f(s) \text{ for } s \in S\}$ is a mixed set of abstract and concrete states, where f is a function such that

$$f(s) = \begin{cases} s & V_{pc}(s) \notin PC_c \\ \alpha_\Phi(s) & \text{otherwise} \end{cases}$$

- x_0 is a start state, so that if $V_{pc}(s_0) \in PC_c$ then $x_0 = \alpha_\Phi(s_0)$, else $x_0 = s_0$,
- $T_x \subseteq S_x \times S_x$ is a set of transitions. More specifically, it consists of concrete transitions $T_x^{c \rightarrow c}$, abstract transitions $T_x^{\alpha \rightarrow \alpha}$, transitions between concrete and abstract states $T_x^{c \rightarrow \alpha}$, and transitions between abstract and concrete states $T_x^{\alpha \rightarrow c}$, more formally

$$\begin{aligned} - T_x^{c \rightarrow c} &= \{(s_i, s_j) \mid (s_i, s_j) \in T \text{ and } V_{pc}(s_i) \notin PC_c \text{ and } V_{pc}(s_j) \notin PC_c\} \\ - T_x^{\alpha \rightarrow \alpha} &= \{(\alpha_\Phi(s_i), \alpha_\Phi(s_j)) \mid (s_i, s_j) \in T \text{ and } V_{pc}(s_i) \in PC_c \text{ and } V_{pc}(s_j) \in PC_c\} \\ - T_x^{c \rightarrow \alpha} &= \{(s_i, \alpha_\Phi(s_j)) \mid (s_i, s_j) \in T \text{ and } V_{pc}(s_i) \notin PC_c \text{ and } V_{pc}(s_j) \in PC_c\} \end{aligned}$$

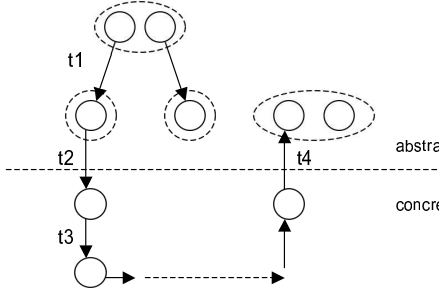


Figure 1: Transitions can occur between the abstract and concrete regions of the

$$- T_x^{\alpha \rightarrow c} = \{(\alpha_{\Phi}(s_i), s_j) \mid (s_i, s_j) \in T \text{ a } PC_c \text{ and } V_{pc}(s_j) \notin PC_c\}$$

- $L_x : S_x \rightarrow 2^{AP}$ is a labeling function, w/ $\{p \in AP \mid x \models p\}$

Figure 1 illustrates the different kinds of transition in a mixed model. In this figure, dotted circles indicate abstract states, solid circles indicate concrete states, and arrows indicate series of transitions, and solid arrows indicate a single transition. Note that in some cases, several concrete states can correspond to the same abstract state. This can happen, for example, when two concrete states at the same level satisfy the same predicates. The dotted line represents the boundary between the concrete and abstract states. In Figure 1, $t_1 \in T_x^{\alpha \rightarrow \alpha}$, $t_2 \in T_x^{\alpha \rightarrow c}$, $t_3 \in T_x^{c \rightarrow c}$, and

We write $s \xrightarrow{t} s'$ to indicate that there is a transition between states s and s' . We write $s \rightarrow s'$ to denote a transition between s and s' when the context is obvious. We say s reaches s' through zero or more transitions, we write $s \rightarrow^* s'$ and say s' is reachable from s .

Path A path $\pi = s_0 \dots s_n$ in a mixed model is a sequence of states such that $(s_i, s_{i+1}) \in T_x$ for $0 \leq i < n-1$. We use $\pi_{s_i}^{s_n}$ to denote that there is a path from s_i to s_n . A mixed path is a path

$$x \xrightarrow{\alpha c^+ \alpha} c = x \xrightarrow{t^{\alpha \rightarrow c}} s_1 \xrightarrow{(t^{c \rightarrow c})^+} s_n \xrightarrow{t^{c \rightarrow \alpha}} c$$

where

$$t^{\alpha \rightarrow c} \in T_x^{\alpha \rightarrow c}, t^{c \rightarrow c} \in T_x^{c \rightarrow c}, \text{ and } t^{c \rightarrow \alpha} \in T_x^{c \rightarrow \alpha}$$

and each of the $t^{c \rightarrow c}$ transitions may be different. Note that in the above mixed path, all states along the path π_x^c remain concrete since the transitions between x and c lie outside the component under test. For two paths $\pi_{x_0}^{x_n}$ and $\pi_{y_0}^{y_n}$, $\pi_{x_0}^{x_n} = \pi_{y_0}^{y_n}$ indicates that these two paths follow the same transitions, and $\pi_{x_0}^{x_n} \neq \pi_{y_0}^{y_n}$ indicates that one or more transitions in each path are different.

Properties The properties are defined only on the abstract part of the model. The concrete part is treated as a single transition. Given a system, CTL* properties will be defined over a simplified system where $x \xrightarrow{\alpha c^+ \alpha} c$ is replaced by a

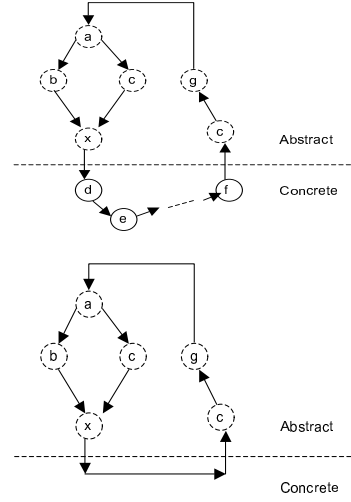


Figure 2: A Kripke structure containing both abstract and concrete states together and the corresponding Kripke structure that will be used in verification.

single transition $x \rightarrow c$. We refer to $x \rightarrow c$ as a border transition.

This is illustrated in Figure 2. The graph on the left side of Figure 2 represents the original Kripke structure for a program and graph on the right side shows the Kripke structure on which properties are defined. As before, dotted circles represent abstract states and solid circles represent concrete states.

Checking the exactness of the abstraction of the transition relation is the biggest challenge we face using mixed abstract and concrete models. If each transition between abstract states is exact with respect to the corresponding transition between concrete states, then a bisimulation relation can be established between the abstract and concrete models and CTL* properties will be preserved in the abstract model. When the transition resides entirely in the abstract part of the system, this check is done easily using the approach presented in [?]. However, the check is less obvious when we must check the exactness of a border transition in which a series of concrete transitions are treated as a single transition. In order to explore all possible behaviors in such border cases, we need to reason about the accumulative effect of the intermediate transitions on the border states. We will use a method similar to symbolic simulation to capture the meaning rather than just the effect of concrete transitions on the border states.

Symbolic transitions A symbolic transition is an accumulation of the effects of a sequence of transitions on the state variables. For example, if we have a series of transitions $x = x + 1$, $x = 2x$, and $x = x^2$ in a path, then symbolic transition for these transitions is $x = (2(x + 1))^2$. Given a path $\pi_{x_0}^{x_n}$, the notation (x_0, x_n) denotes the symbolic transition

sition from x_0 to x_n . In other words,

$$(x_0, x_n) = (x_0, x_1) \circ (x_1, x_2) \circ \dots \circ (x_{n-1}, x_n)$$

where

$$0 \leq i \leq n - 1 \text{ and } (x_i, x_{i+1}) \in T_x$$

and transition composition, \circ , denotes the sequential application of a transition to the result of the previous transition.

As in [?], the precision check is based on weakest preconditions. However, we must cope with weakest preconditions defined over border transitions as well as component transitions. In either case, the basic definition is the same.

Weakest Precondition The weakest precondition, in terms of a transition, calculates the predicates that must be satisfied before a transition executes so that a set of predicates will be satisfied after the transition. We use $wp(\Phi, i)$ to express weakest precondition of transition i in terms of a predicate set Φ . We have $wp(\Phi, i) = \Phi[x_{new}/x_{old}]$ where x_{new} is the new value of variable x in the predicate set Φ after a transition i , and x_{old} is the old value of variable x before the transition i .

2. THEOREMS

The central problem in our mixed computational model is determining whether or not the abstraction is precise, or, in other words, creates a bisimulation between the partially abstract and the original systems. And this problem is particularly difficult for border transitions which exit and re-enter the abstracted component. In this section, we provide theorems that characterize the correctness and termination properties of the algorithm in the presence of border transitions.

There are several scenarios in which the abstraction of the component may lose precision after execution passes through then returns from the concrete environment. Figure 3 summarizes these cases when execution re-enters the abstract component. Figure 4 illustrates the cases when execution fails to re-enter the abstract component. All other possible behaviors are special cases of these four behaviors. In these figures, solid circles indicate concrete states, dotted circles indicate abstract states, solid arrows indicate one transition, and dotted arrows indicate series of transitions.

In the left graph of figure 3, border states x and y are abstracted into same abstract state, go through same transitions, and eventually generate two different abstract states. In this case, we need to add predicates to distinguish x and y in order to explore both c and d . We will symbolically simulate the transitions in the concrete environment and check the weakest precondition of the predicates in terms of the final result of symbolic simulation through the concrete environment to decide if the abstraction is exact or not. If the symbolic transition for a sequence of concrete transitions is $x = (2(x + 1))^2$ then to determine if the abstraction is precise, we check if $\alpha_\Phi(x) \Rightarrow wp(\Phi, x = (2(x + 1))^2)$

In the right graph of figure 3, border states x and y go through same transitions for a while, reach states z and w ,

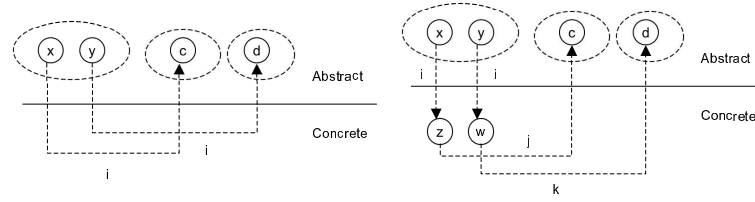


Figure 3: Scenarios in which control flows from a single abstract state into the surrounding software and then back into the abstracted component

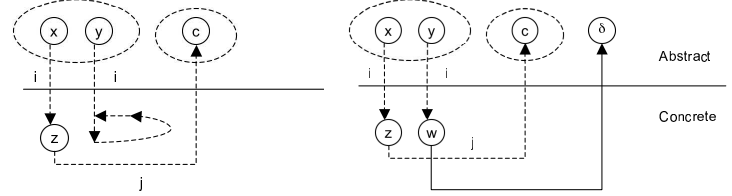


Figure 4: Scenarios in which control flows out of but not back into an abstract component.

then separate into different transitions, and end up with different abstract states when they re-enter the component. This case occurs because there are branches in the concrete environment. In a guarded command language, there needs to be guards that distinguish z and w in the concrete environment. The intuition of our approach is to push the guards that distinguish z and w upwards into the path to generate predicates that distinguish x and y . For example, in this graph, let γ represent the guards that differentiate z and w , then we check if $\alpha_\Phi(x) \Rightarrow wp(\gamma, i)$, where i is the symbolically simulated transition for the series of transitions before the branch occurs.

In the left graph of figure 4, border state y has an infinite loop, and never returns to the abstract component. Detecting loops such as these is equivalent to solving the halting problem. Therefore, if the concrete environment executes infinitely without returning control to the component, we can't say anything about the component. In the right figure, border state y terminates in the concrete environment and never returns to the abstract component. In this case, we introduce an exit state δ in abstract component. Using this placeholder abstract state, the case is similar to the scenarios described in figure 3, and we can use the approaches described there to determine the precision of the abstraction.

Recall that our algorithm does abstraction refinement using Pasareanu's approach inside the component, and does abstraction refinement for border states by checking the exactness of the abstraction in terms of a symbolic representation of the accumulated behavior of a series of concrete transitions and in terms of the guards of branches encountered on border transitions. Abstraction refinement in border states can be done by considering just a prefix of or all of transitions in the concrete environment.

The precision of the abstract component will increase as the number of concrete transitions considered increases. When we consider all the transitions, and the algorithm terminates, then we have an exact abstraction which creates a

bisimulation and can make claims about the correctness of the component. The following theorems precisely state and prove these claims.

Theorem 1 proves that doing abstraction refinement on symbolic representations of sequences of transitions preserves the behaviors for each of the four possible boundary cases. The theorem is split into two cases. The first case corresponds to the left side of figure 3 and the second case corresponds to the right side of 3. The proof uses composition and weakest precondition to show that the algorithm will include all possible abstract behaviors relative to the given properties whether or not there is a branch in the concrete environment.

THEOREM 1. *Let Φ_i , ($0 \leq i \leq n$) denote the set of local predicates in program location of x_i . Given a path $\pi_{x_0}^{x_n}$, and y_0 . such that $\alpha_{\Phi_0}(y_0) = \alpha_{\Phi_0}(x_0)$,*

1. *if each of the following conditions are met:*

$$\begin{aligned} \pi_{x_0}^{x_n} &= \pi_{y_0}^{y_n}, \\ \alpha_{\Phi_n}(x_n) &\neq \alpha_{\Phi_n}(y_n) \text{ and} \\ \exists \Phi'_0. \alpha_{\Phi'_0}(x_0) &\Rightarrow wp(\Phi_n, (x_0, x_n)) \end{aligned}$$

then $\alpha_{\Phi'_0}(x_0) \neq \alpha_{\Phi'_0}(y_0)$

2. *if each of the following conditions are met:*

$$\begin{aligned} x_0 &\xrightarrow{*} x_{i-1} \xrightarrow{t_x} x_i \xrightarrow{*} x_n \\ y_0 &\xrightarrow{*} y_{i-1} \xrightarrow{t_y} y_j \xrightarrow{*} y_m, \\ t_x &\neq t_y, \\ \pi_{x_0}^{x_{i-1}} &= \pi_{y_0}^{y_{i-1}}, \\ \pi_{x_0}^{x_n} &\neq \pi_{y_0}^{y_m}, \text{ and} \\ \exists \Phi'_0. \alpha_{\Phi'_0}(x_0) &\Rightarrow wp(\Phi_{i-1}, (x_0, x_{i-1})) \end{aligned}$$

in which Φ_{i-1} is a set of guards that differentiate x_{i-1} and y_{i-1} , then $\alpha_{\Phi'_0}(x_0) \neq \alpha_{\Phi'_0}(y_0)$.

PROOF. 1. By induction with respect to the number of transitions between x_0 and x_n . Basis step with ($n = 1$) is obvious. Induction step: Assume $n = k + 1$. From the definition of composition and weakest precondition, we know that

$$\alpha_{\Phi'_0}(x_0) \Rightarrow wp(\Phi_{k+1}, (x_0, x_{k+1}))$$

is equivalent to

$$\alpha_{\Phi'_0}(x_0) \Rightarrow wp(wp(\Phi_{k+1}, (x_k, x_{k+1})), (x_0, x_k)).$$

Let $\Phi_k = wp(\Phi_{k+1}, (x_k, x_{k+1}))$, then from the definition of weakest precondition, we have that the abstractions of successor states for states z_k and x_k are equal under a transition t iff the abstractions of z_k and x_k are equal. More precisely,

$$z_k \xrightarrow{t} z_{k+1} \wedge x_k \xrightarrow{t} x_{k+1}$$

implies

$$\alpha_{\Phi_{k+1}}(z_{k+1}) = \alpha_{\Phi_{k+1}}(x_{k+1}) \text{ iff } \alpha_{\Phi_k}(z_k) = \alpha_{\Phi_k}(x_k).$$

However, we assumed that $\alpha_{\Phi_{k+1}}(x_{k+1}) \neq \alpha_{\Phi_{k+1}}(y_{k+1})$ and $(x_0, x_{k+1}) = (y_0, y_{k+1})$, from which it follows that

$\alpha_{\Phi_k}(x_k) \neq \alpha_{\Phi_k}(y_k)$, and $\alpha_{\Phi'_0}(x_0) \neq \alpha_{\Phi'_0}(y_0)$ follows from the inductive hypothesis.

2. Proof is similar to part 1. Since Φ_{i-1} is a set of guards that distinguish x_{i-1} and y_{i-1} , we have that $\alpha_{\Phi_{i-1}}(x_{i-1}) \neq \alpha_{\Phi_{i-1}}(y_{i-1})$. We assumed $\pi_{x_0}^{x_{i-1}} = \pi_{y_0}^{y_{i-1}}$, $\alpha_{\Phi_0}(x_0) = \alpha_{\Phi_0}(y_0)$, and $\alpha_{\Phi'_0}(x_0) \Rightarrow wp(\Phi_{i-1}, (x_0, x_{i-1}))$. The same use of weakest precondition in part 1 gives $\alpha_{\Phi'_0}(x_0) \neq \alpha_{\Phi'_0}(y_0)$.

□

We have the following corollary of the above theorem. The corollary says that if the abstraction is exact in terms of the symbolically simulated transitions which lie outside the component, then the abstraction includes all the possible behaviors of the states at the border of the component.

COROLLARY 1. *Given a series of transitions $x_0 \rightarrow x_1 \rightarrow \dots \rightarrow x_n$, if the abstraction is exact for symbolically represented transition (x_0, x_n) using a set of predicates Φ_n , then the abstraction captures all effects of concrete behaviors starting at location 0 at program location n in terms of the predicates Φ_n .*

PROOF. This corollary is a direct result of theorem 1. Suppose $\alpha_{\Phi_0}(x_0)$ is an abstract state produced in program location 0, and $\alpha_{\Phi_n}(x_n)$ is an abstract state produced in program location n . The concrete transition $x_0 \rightarrow x_1 \rightarrow \dots \rightarrow x_n$ will be replaced by symbolic transition (x_0, x_n) . States x_0 and x_n can be abstracted into existing abstract states or new abstract states. There are four different possibilities. First, both $\alpha_{\Phi_0}(x_0)$ and $\alpha_{\Phi_n}(x_n)$ are existing abstract states. Second, $\alpha_{\Phi_0}(x_0)$ is an existing abstract state and $\alpha_{\Phi_n}(x_n)$ is a new abstract state. Third, $\alpha_{\Phi_0}(x_0)$ is a new abstract state and $\alpha_{\Phi_n}(x_n)$ is an existing abstract state. Fourth, both $\alpha_{\Phi_0}(x_0)$ and $\alpha_{\Phi_n}(x_n)$ are new abstract states. In the first case, since both states are existing states, no precision is lost. The second case is impossible by theorem 1. In the third case, the new state is generated in program location 0 and the result of execution is an existing state at program location n , which does not lose precision. In the fourth case, a new abstract state is generated in program location 0, which goes to a new abstract state at program location n , which also does not lose precision. Therefore, the corollary holds for all possible cases. □

The next theorem states we can vary the precision of the abstraction by varying the number of concrete transitions used to build the symbolic representation of the surrounding software. In all cases, errors found in the abstract component correspond to feasible errors in the original component.

THEOREM 2. *Suppose $x_0 \xrightarrow{\alpha_c^+ \alpha} x_n$, then the abstract component generated by doing abstraction refinement in terms of (x_0, x_i) for the predicate set Φ_i , with $0 < i < n$ and $\Phi_i = AP$, is an under-approximation of the original component.*

PROOF. Since all states and transitions visited during model checking are concrete states and generated using concrete

transitions from concrete software, all errors found during model checking correspond to real errors. Since $0 < i < n$, the exactness check will include concrete behaviors up to the instruction at program location i . However, program behaviors between the instructions at locations i and n will be ignored in the precision check. This means that some execution paths that split between locations i and n may be missed. If these paths lead to errors in the component, then those errors will not be detected. Therefore, the absence of errors in the abstract system does not guarantee the absence of errors in the concrete system. \square

This following theorem establishes the relationship between sets of errors found when using symbolic representations built from different numbers of steps through the concrete environment.

THEOREM 3. *Suppose $x_0 \xrightarrow{\alpha c^+ \alpha} x_n$, and suppose E_i is the set of errors detected in the abstract component by doing refinement in terms of (x_0, x_i) , and E_{i+1} is the set of errors detected in the abstract component by doing abstraction refinement in terms of (x_0, x_{i+1}) , where $i < i + 1 < n$, then,*

$$E_i \subseteq E_i \cup E_{i+1} \text{ and } \bigcup_0^n E_i = E_n \text{ but not } E_i \subseteq E_{i+1}$$

PROOF. $E_i \subseteq E_i \cup E_{i+1}$ is obvious. However, it is not the case that $E_i \subseteq E_{i+1}$. For example, if the accumulative transition through i steps is $x = x - 1$ and the next transition is $x = x + 1$, then the accumulative transition through $i + 1$ steps is $x = x$. If the complete accumulative transition is $x = x - 1$, then the model of i steps is more precise than the model of $i + 1$ steps. Since the accumulative transition (x_0, x_n) is the only complete model of concrete program behavior, the errors found in the n th step include all the errors. \square

We have dealt with transitions that enter and leave the component in the previous theorems. We now have the foundation on which to describe our model in terms of the one described in [?] and give a slightly modified version of the theorems presented in [?].

Theorem 4 states that if the algorithm terminates, we either find an error or prove that the system is error-free.

THEOREM 4. *$\forall x_0. x_0 \xrightarrow{\alpha c^+ \alpha} x_n$, if we do abstraction refinement in terms of (x_0, x_n) and the algorithm terminates with errors, then the errors are real. If the algorithm terminates without any error, then the component is error free.*

PROOF. The algorithm terminates only when an error is found or the abstraction is exact with respect to all symbolically represented transitions. As shown in theorem 2, if an error is found, it will be a feasible error because we only explore real concrete states and real concrete transitions. And from corollary 1, we know that we can treat complete symbolically simulated transitions through the concrete software as a single transition because the abstraction does not lose any precision relative to such transitions for paths which start at the start location and end at the end location. Also,

behaviors from the concrete software are ignored during verification. If we treat each of the symbolically simulated transitions as one single transition, then we will get a system that is presented in [?] and therefore all of the theorems presented there also apply to our system. \square

Theorem 5 describes the cases in which the abstraction computation may not terminate.

THEOREM 5. *If the algorithm does not terminate, there are two possibilities.*

1. *The concrete system includes infinite behaviors in which case we don't know anything about the component, or,*
2. *if the algorithm does not terminate because of infinite behavior in the component, then the algorithm will eventually generate a structure which is bisimilar to the original component although the algorithm is not able to detect such a bisimulation.*

PROOF. 1. The first statement holds due to fundamental limits of computability. In particular, determining if the concrete software terminates or not is the halting problem.

2. The theorem in [?] applies directly.

\square