

FORMAL MODELING OF PROCESS MIGRATION

Aric D. Blumer[†], Henning Mortveit[‡], and Cameron D. Patterson[†]

¹Bradley Department of Electrical and Computer Engineering

²Virginia Bioinformatics Institute

Virginia Polytechnic Institute and State University

Blacksburg, VA 24061

email: aric@vt.edu, henning.mortveit@vt.edu, cdp@vt.edu

ABSTRACT

This paper develops a formal model of process migration that describes programs, processes, and the migration of those processes within a migration realm. A migration realm is a group of processors modeled as finite state machines. The model is motivated by a migration application between software and Field Programmable Gate Array (FPGA) hardware, and the theorems of the model guide the use of FPGA resources while guaranteeing complete and correct execution of a process. By defining different types of migration realms this paper also develops a migration realm taxonomy.

1. INTRODUCTION

The goal of this paper is to present a formal model that describes in general the migration of processes among processors. It defines a *program* and a *process*, which is an instance of a program. A process is migrated within a *migration realm*, which is a collection of computing devices modeled in terms of Finite State Machines (FSMs). The instructions of a process are divided into sequences that are serially executed on one or more processors within the realm until the process completes. A process exists in an abstract form that can be translated as needed into a form used by concrete processors. This model also forms the basis for a migration system taxonomy, and its theorems present a set of criteria which guarantees the correct completion of programs in a set of heterogeneous processors.

While the model is generally applicable to many forms of migration systems, its impetus is the migration of processes between software and FPGA hardware. Run-time reconfiguration of FPGAs provides the means of implementing a migration realm within an FPGA, and one application is the acceleration of digital circuit simulation. FSMs were chosen as the basis of the model primarily because digital circuits are often described in the form of FSMs, so the migration of FSM state is natural in this context. However, the

[†]This work was graciously funded through a Virginia Tech College of Engineering Fellowship and a Bradley Fellowship.

model defines its components in general enough terms to be applicable to Turing Machines as well.

Section 2 presents related work, and Section 3 establishes the foundational definitions of strings and FSMs referred to in subsequent sections. Section 4 describes a migration realm, a program, and a process. Section 5 elaborates on how to simplify migration realms by requiring certain similarities between FSMs. Section 6 describes an application to run-time reconfiguration of FPGAs in light of this model, followed by Section 7 which gives an overview of future work and the conclusion.

2. RELATED WORK

Brand and Zafiropolu, in the foundational work on communicating finite state machines, model FSM communication through a *protocol* [1]. The object is to detect missing or “unexecutable” messages. Pétrot *et al.* have modeled digital simulation with FSMs [2], but this model is essentially a communicating FSM problem rather than state migration. Similarly, Milner *et al.* have developed the π -calculus [3], which describes a set of communicating mobile processes that can change their communication link structure. This paper describes the migration of machine state without consideration of communication with other machines. Communication between collaborating FSMs is orthogonal to the migration of each machine’s state.

Migration has also been studied in *ad-hoc* networks in projects such as MobileUNITY [4], but such work focuses on sharing, synchronization, and security of mobile computing systems rather than the state migration itself. More closely related to our work is the MAIL [5] language which formalizes itineraries of a process through a network, but it does not address translation of code and state. It also focuses upon autonomous mobile agents rather than passive processes, the object of our work. Operating system process migration has also been studied extensively [6], but this type of migration is an application at a higher level than that addressed here.

3. FOUNDATIONAL DEFINITIONS

A program is modeled as a string of input symbols to an FSM, and there must be a means to represent which symbols have been processed and which have not. A *string*, then, is a sequence of symbols such as *abcdef*, and a *substring* is a string that forms a part of another string. For example, *cde* is a substring of *abcdef*. The empty string is ϵ , and the set of all strings over set A is denoted A^* . The elements of a string are identified using zero-based subscripts, so if $\sigma = abcdef$, $\sigma_0 = a$. The length of string σ is denoted $|\sigma|$. The *concatenation* of two strings is the juxtaposition of each sequence of symbols from each string. A string σ concatenated with ϵ is σ , so ϵ is the identity under concatenation.

A Finite State Machine (FSM), also called a finite automaton, provides the basis upon which strings and substrings are applied to establish the notion of a program and a process. An FSM is a machine that processes input strings, changing its state for each symbol, and it can be classified as either an acceptor or a transducer, depending on what the output of the FSM is considered to be. FSMs are often described with directed graphs whose vertices are the states, and whose arcs are the state transitions labeled with the symbol causing the transition, such as that shown in Fig. 1. It shows a simple FSM that processes a series of input symbols and determines if they spell “computer.” Arithmetic computation can be represented with an FSM, but the number of states easily becomes large. For example, a machine with twelve storage bits has a state space of 4,096 states. To limit the examples in this paper to a reasonable number of states, only simple FSMs are used in the examples, but the principles apply for computational machines with much larger state spaces.

Migration involves the states of FSMs rather than the output of those FSMs, and the initial states become a property of “programs” rather than the machines themselves. Therefore, we use semiautomata, which are reduced FSMs that omit initial states and anything related to output such as acceptor states and output functions. For the remainder of this paper, we use the term FSM as a synonym for a reduced FSM, unless stated otherwise.

Definition 1. A *reduced finite state machine* is a triple

$$\mathcal{A} = (S, A, \delta) \quad (1)$$

where S is a finite set, and $\delta: S \times A^* \rightarrow S$ is a function such that for all $s \in S$, $a \in A$, and $\sigma \in A^*$,

$$\delta(s, \epsilon) = s \quad \text{and} \quad (2)$$

$$\delta(s, \sigma a) = \delta(\delta(s, \sigma), a). \quad (3)$$

The set S is the set of *states*, A is the *input alphabet*, and the elements of σ and symbol a are the *input symbols*. The function δ is the *transition function* of the FSM. \square

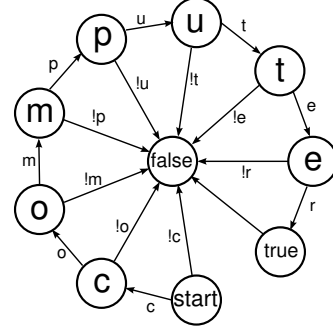


Fig. 1. An acceptor FSM recognizing “computer”

For the remainder of this paper, when an FSM \mathcal{A} is referenced, S , A , and δ are assumed to be the elements of \mathcal{A} , though not explicitly mentioned. Similarly, a reference to \mathcal{A}_i infers S_i , A_i , and δ_i .

4. MIGRATION REALMS AND PROCESSES

The migration of a process requires a source processor and a destination processor, so any group of processors can provide a “realm” within which a process may migrate. Intuitively, if there is a set of processors that are identical in every respect, the migration of state is direct. But what if there is a need to migrate a process between two processors that use different instruction sets or have different numbers of registers? Such differences are shown in this model using unequal alphabets, states, and transition functions.

Definition 2. A *migration realm* \mathcal{R} is a set of FSMs

$$\mathcal{R} = \{\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2, \dots\}. \quad (4)$$

$|\mathcal{R}|$ is the number of FSMs in the realm. A *homogeneous migration realm* \mathcal{R} is a migration realm such that for all $\mathcal{A} \in \mathcal{R}$

$$S = S', A = A', \text{ and } \delta = \delta', \quad (5)$$

for arbitrary S' , A' , and δ' . Otherwise, the migration realm is *heterogeneous*. \square

Within a realm a *process*, which is an instance of a *program*, is executed and migrated.

Definition 3. A *program* $P = (\mathcal{A}, s_0, \sigma)$ is a triple consisting of a finite state machine $\mathcal{A} = (S, A, \delta)$, an initial state $s_0 \in S$, and a string of input symbols σ such that $A = \{\sigma_0, \dots, \sigma_{|\sigma|-1}\}$. A *process* p of program P is a quadruple $p = (\mathcal{A}, s, \sigma, c)$ where

$$s = \begin{cases} s_0, & \text{for } c = 0 \\ \delta(s_0, \sigma_{[0, c-1]}), & \text{for } 0 < c \leq |\sigma|. \end{cases} \quad (6)$$

s is the *current state*, and c is the *input counter*. The process is *complete* when $c = |\sigma|$. \square

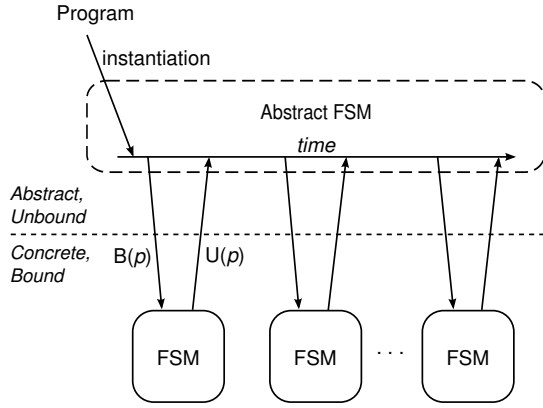


Fig. 2. Process migration through multiple concrete FSMs

Each machine in a realm is not required to be able to process an entire program, so Def. 3 incorporates an abstract FSM into a program to serve as a reference. This abstract machine is guaranteed by definition to be able to execute the entire program, but it may not have a complete implementation in the realm. The program itself is also abstract. It has not begun execution, and it can be instantiated as multiple processes. In later definitions, the superscribed hat notation (e.g., $\hat{\mathcal{A}}$) identifies abstract elements to differentiate between abstract and concrete elements. Computer source code is compiled into sections such as “text” and “data.” The text is the instructions, analogous to an input string σ in this model, and the data section is the initial state of the program, analogous to s_0 in this model.

A process is a program instance, and the input counter tells how far into the program the process has executed to arrive at the current state. A process is still abstract because it exists in terms of its program’s abstract FSM. That is, it is not necessarily in a form that can be executed on a concrete FSM, so the superscribed hat notation is employed to identify it as such when the context requires. The flow of a process from its abstract form to concrete forms is illustrated in Fig. 2. Through “binding,” shown as $B(p)$, the process is translated into a form that a concrete FSM can process. When the FSM has processed its portion of the program, it is unbound ($U(p)$), or translated back into terms of the abstract FSM. So a *process* is abstract, and a *bound process* is concrete. A binding function is the means of producing a bound process, and there are binding functions for the input string and for state. The binding of the state is straightforward. Determining if the input string can be bound, however, is a more difficult problem since the target FSM is not required to process all of the remaining input string.

Definition 4. Binding, Unbinding, and Migration

a) Let σ be a string. σ is a *bindable string* to \mathcal{A} if there exists an injective function $B_A : \{\sigma\} \rightarrow A^*$ such that $B_A(\sigma) \neq \epsilon$. The image $B_A(\sigma)$ is the *bound string*.

b) Let $\hat{p} = (\hat{\mathcal{A}}, \hat{s}, \hat{\sigma}, \hat{c})$ be a process of program $\hat{P} =$

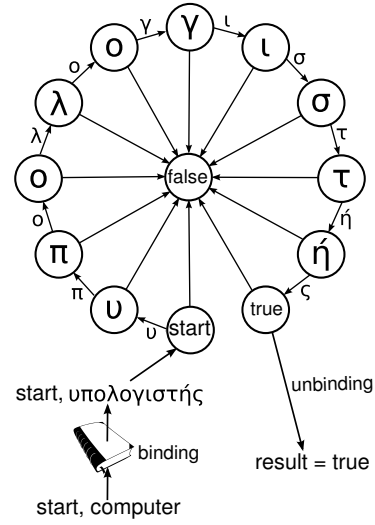


Fig. 3. Migration through a Greek acceptor FSM

$(\hat{\mathcal{A}}, \hat{s}_0, \hat{\sigma})$. The six-tuple $p = (\hat{\mathcal{A}}, \hat{\sigma}, \hat{k}, \mathcal{A}, s, \sigma)$ is a *bound process* of \hat{p} to \mathcal{A} if there exists a bindable string $\hat{\tau} = \hat{\sigma}_{[c, \hat{k}-1]}$ such that $\sigma = B_A(\hat{\tau})$, and there exists a function $B_S : \{\hat{s}\} \rightarrow S$ such that $s = B_S(\hat{s})$. \hat{k} is the input counter marking the end of the bindable string. The relationship between \hat{p} and p is denoted $p = B(\hat{p})$. If B_S and σ exist, then both \hat{p} and \hat{P} are *bindable* to \mathcal{A} . Process \hat{p}' is the *unbound process* of p if $\hat{p}' = (\hat{\mathcal{A}}, \hat{s}', \hat{\sigma}, \hat{k})$ such that $\hat{s}' = U_S(\hat{\sigma}(s, \sigma)) = \hat{\delta}(\hat{s}, \hat{\tau})$. The relationship between \hat{p}' and p is denoted $\hat{p}' = U(p)$.

c) Function $M : \{p\} \rightarrow \{p\}$ is a *migration function* such that $p' = M(p) = B(U(p))$. M_{ij} denotes the migration function from \mathcal{A}_i to \mathcal{A}_j . □

For the sake of brevity, further references to a process \hat{p} or a program \hat{P} infer the constituents $\hat{\mathcal{A}}$, \hat{s}_0 , \hat{s} , $\hat{\sigma}$, and \hat{c} .

Example. The program of Fig. 1 is defined as $\hat{P} = (\hat{\mathcal{A}}, \text{start}, \text{computer})$, where $\hat{\mathcal{A}}$ contains the states and transition function shown, with an input alphabet of the 26 English letters. Suppose that a migration realm consists of the single FSM shown in Fig. 3 with an input alphabet of the 24 Greek letters (ignoring diacritical marks). The state binding function B_S maps the English FSM *start* state to the Greek FSM *start*. The unbinding function U_S maps the Greek FSM *true* to the English FSM *true*, and it maps all other Greek FSM states to *false* of the English FSM. The input string binding function B_A is an English-to-Greek dictionary. When the input string *computer* is translated to *υπολογιστής*, the concrete FSM in Fig. 3 processes it, and the final unbound state is *true*. Through this sequence, the program is executed in a completely different FSM from which it was defined, but the result is the same.

The definition of an unbound process (Def. 4b) also addresses correctness. A process may be bindable to a concrete FSM, but that does not guarantee that the final state may be mapped back to the abstract FSM’s states, nor does it guarantee that it is correct. The unbinding requirement that

$U_S(\delta(s, \sigma)) = \hat{\delta}(\hat{s}, \hat{\tau})$ ensures that the result is correct. That is, if the final state of the bound process—when mapped back into the abstract state space—is the same as the state that the abstract FSM produces, then the final state is correct. Otherwise the process is stranded. It is a waste of resources in an implementation to strand processes, so execution to completion must be ensured. We must guarantee that the realm can complete a process before it begins running.

Definition 5. Let \hat{p} be a process of program \hat{P} . \hat{P} is *realm executable* in \mathcal{R} if there exists a finite sequence of bound processes $\mathbb{P} = (p_1, p_2, \dots, p_n)$ such that

$$p_{i+1} = \begin{cases} B(\hat{p}), & \text{for } i = 0, \\ M(p_i), & \text{for } 0 < i < n, \end{cases} \quad (7)$$

and there exists \hat{p}' such that $\hat{p}' = U(p_n)$. The sequence of ordered pairs of FSM and bindable string from \mathbb{P} is the *execution path*. The sequence of FSMs from \mathbb{P} is the *FSM path*. \square

Theorem 1. Given a homogeneous migration realm $\mathcal{R} = \{\mathcal{A}_1, \dots, \mathcal{A}_n\}$, a program \hat{P} that is realm executable in \mathcal{R} is also realm executable in any non-empty subset of \mathcal{R} .

Proof. Let $\mathbb{A} = (\mathcal{A}_1, \dots, \mathcal{A}_n)$ be an arbitrary FSM path of \hat{P} . Let $\mathcal{R}' = \{\mathcal{A}_i\}$, a non-empty subset of \mathcal{R} . Because \mathcal{R} is homogeneous, for all $\mathcal{A} \in \mathcal{R}$, $\mathcal{A} = \mathcal{A}_i$, so $\mathbb{A} = (\mathcal{A}_i, \dots, \mathcal{A}_i)$. Therefore, \hat{P} is realm executable in \mathcal{R}' . \square

5. REDUCING HETEROGENEITY

Def. 2 defines a heterogeneous migration realm as a realm in which any of the three elements of the FSM triple differ between automata. Definitions 4 and 5 formalize the requirements that must be satisfied in order for a program to be realm executable. These are general requirements for all realms, and the correctness requirement of Def. 4— $U_S(\delta(s, \sigma)) = \hat{\delta}(\hat{s}, \hat{\tau})$ —implies that the program must be processed by the abstract machine's transition function ($\hat{\delta}$) in order to know for certain if the result is correct. This requirement defeats the purpose of having a realm. However, a close examination of Def. 4 shows that the requirements depend on two sets of functions: the binding/unbinding functions (B_S , B_A , and U_S), and the state transition functions (δ and $\hat{\delta}$). The difficulty of satisfying the correctness requirement can be reduced by applying constraints that simplify the binding functions.

Let us constrain the state binding and unbinding functions to be identity functions. In other words, the machines use the same states and alphabets. Now, the correctness criterion from Def. 4 is summed up as follows: $U_S(\delta(B_S(\hat{s}), B_A(\hat{\tau}))) = \hat{\delta}(\hat{s}, \hat{\tau})$. When using the identity binding functions, this criterion is simplified significantly: $\delta(\hat{s}, \hat{\tau}) = \hat{\delta}(\hat{s}, \hat{\tau})$. But this equality, by Def. 1, can only be

true if \hat{s} and $\hat{\tau}$ are elements of the domains of both δ and $\hat{\delta}$. That is, $\hat{s} \in (\hat{S} \cap S)$ and $\hat{\tau} \in (\hat{A} \cap A)^*$. So correctness is guaranteed if δ and $\hat{\delta}$ are equivalent when limiting their domains to $(S \cap \hat{S}) \times (A \cap \hat{A})^*$. Limiting the domain of a function to a subset of its defined domain is called a restriction, and it is denoted, in this case, $\delta|_{(S \cap \hat{S}) \times (A \cap \hat{A})^*}$. There are several relationships between S and \hat{S} and between A and \hat{A} that can satisfy this correctness criterion, and we investigate them in Def. 6.

Definition 6. A migration realm \mathcal{R} is a *restricted realm* of program \hat{P} if for all $\mathcal{A} \in \mathcal{R}$,

$$\delta|_{(S \cap \hat{S}) \times (A \cap \hat{A})^*} = \hat{\delta}|_{(S \cap \hat{S}) \times (A \cap \hat{A})^*},$$

and B_S , B_A , and U_S are the identity functions. A realm \mathcal{R} of \hat{P} is a *state-equivalent* (SE) realm if for all $\mathcal{A} \in \mathcal{R}$, $S = \hat{S}$. Consider the following three constraints for FSMs of a state-equivalent restricted realm \mathcal{R} of \hat{P} :

1. $A = \hat{A}$ (identity machine)
2. $(A \cap \hat{A}) \subsetneq \hat{A}$ (SE subset machine)
3. $A \supset \hat{A}$ (SE superset machine)

If for all $\mathcal{A} \in \mathcal{R}$, constraint 1 is true, then \mathcal{R} is an *identity realm*. If for all $\mathcal{A} \in \mathcal{R}$, constraint 2 is true, then \mathcal{R} is an *SE subset realm*. If for all $\mathcal{A} \in \mathcal{R}$, constraint 3 is true, then \mathcal{R} is an *SE superset realm*. \square

Note the state-equivalent realm defined above. It is ubiquitous in computing systems that state is stored in binary digits. While there are differences in byte order and floating point representation, we assume that mappings between them are isomorphic, so state equivalency is a reasonable simplification. The actual migration system that is motivating the development of this model consists of a set of virtual machines run in software and real machines run in hardware, and they share identical instruction sets and memories. They form an SE realm. We therefore focus on SE realms for the remainder of this model, but future work includes the study of restricted realms that are not state equivalent, a much harder problem. The migration system implementation can be extended to support the compilation of processes directly to hardware which will not necessarily be state equivalent, and so we must investigate how to guarantee correct completion.

Theorem 2. For every bound process in an SE restricted realm, there exists an unbound process.

Proof. Let $p = (\hat{\mathcal{A}}, \hat{\sigma}, \hat{k}, \mathcal{A}, s, \sigma)$ be a bound process in SE restricted realm \mathcal{R} . Since B_A is the identity function, σ is both the bindable string and the bound string. Let $\hat{p} = (\hat{\mathcal{A}}, \hat{s}, \hat{\sigma}, \hat{k})$ be a process. From Def. 4, \hat{p} is an unbound process of p if $\hat{s} = \delta(s, \sigma) = \hat{\delta}(s, \sigma)$, since U_S is the identity function. Since $\delta|_{(S \cap \hat{S}) \times (A \cap \hat{A})^*} = \hat{\delta}|_{(S \cap \hat{S}) \times (A \cap \hat{A})^*}$ for a restricted realm, it must be shown that $s \in (S \cap \hat{S})$ and $\sigma \in (A \cap \hat{A})^*$.

Since σ is the bound string, then by Def. 3a, $\sigma \in A^*$. Since σ is also the bindable string, $\sigma \in \hat{A}^*$. Thus, $\sigma \in (A^* \cap \hat{A}^*)$. Therefore, $\sigma \in (A \cap \hat{A})^*$.

The definition of a bound process indicates that $s \in S$, and for a state equivalent realm, $S = \hat{S}$. Hence, $s \in \hat{S}$. Consequently, $\delta(s, \sigma) = \hat{\delta}(s, \sigma)$. Therefore, there exists an unbound process for every bound process in an SE restricted realm. \square

Theorem 3. *Given program \hat{P} , let $\mathcal{R} = \{\mathcal{A}_1, \dots, \mathcal{A}_n\}$ be an SE restricted realm of \hat{P} . \hat{P} is realm executable in \mathcal{R} if $\hat{A} \subseteq A_1 \cup A_2 \cup \dots \cup A_n$.*

Proof. For this proof, a sequence of bound processes is constructed that meet the requirements of Def. 5. The sequence contains one bound process for each input symbol.

I. It must be shown that for every input symbol there exists a machine to which it is bindable. Let $\mathbb{A} = A_1 \cup A_2 \cup \dots \cup A_n$. The definition of a program shows that for all $\hat{\tau} \in \hat{\sigma}$, $\hat{\tau} \in \hat{A}$. Since $\hat{A} \subseteq \mathbb{A}$, for all $\hat{\tau} \in \hat{\sigma}$, $\hat{\tau} \in \mathbb{A}$. Therefore, for every symbol in $\hat{\sigma}$ there exists a machine to which it is a bindable string.

II. Because the realm is state-equivalent, $S = \hat{S}$. Because the realm is restricted, B_S and U_S exist and they are the identity functions.

III. Now let us show that for every adjacent pair of bound processes, one can migrate from the first to the second. Let

$$S_i = \begin{cases} s_0, & \text{for } i = 0 \\ \hat{\delta}(s_0, \hat{\sigma}_{[0, i-1]}), & \text{for } 0 < i \leq |\hat{\sigma}|. \end{cases} \quad (8)$$

Let \mathcal{A}_i be a machine to which symbol $\hat{\sigma}_i$ is bindable. Let bound process $p_i = (\hat{\mathcal{A}}, \hat{\sigma}, i + 1, \mathcal{A}_i, S_i, \hat{\sigma}_i)$. Let $\mathbb{P} = (p_0, p_1, \dots, p_{|\hat{\sigma}|-1})$ be a sequence of bound processes, one per input symbol. To satisfy Def. 5, it must be proven that $p_{i+1} = M(p_i)$, for $0 < i < |\hat{\sigma}| - 1$. That is, it must be proven that there exists a process \hat{p} such that \hat{p} is the unbound process of p_i , and p_{i+1} is a bound process of \hat{p} . Theorem 2 indicates that $\hat{p}_i = (\hat{\mathcal{A}}, \hat{\delta}(S_i, \hat{\sigma}_i), \hat{\sigma}, i + 1)$ exists as the unbound process of bound process p_i . Next is proven that p_{i+1} is a bound process of \hat{p}_i . As defined above for \mathbb{P} , $p_{i+1} = (\hat{\mathcal{A}}, \hat{\sigma}, (i + 1) + 1, \mathcal{A}_{i+1}, S_{i+1}, \hat{\sigma}_{i+1})$. From Def. 4, the bound form of \hat{p}_i is $(\hat{\mathcal{A}}, \hat{\sigma}, (i + 1) + 1, \mathcal{A}_{i+1}, \hat{\delta}(S_i, \hat{\sigma}_i), \hat{\sigma}_{i+1})$. These are identical except for the state.

$$\begin{aligned} S_{i+1} &= \hat{\delta}(S_i, \hat{\sigma}_i) \\ &= \hat{\delta}(\hat{\delta}(s_0, \hat{\sigma}_{[0, i-1]}), \hat{\sigma}_i) && \text{(Def. of } S_i) \\ &= \hat{\delta}(s_0, \hat{\sigma}_{[0, i-1]} \hat{\sigma}_i) && \text{(Def. 1)} \\ &= \hat{\delta}(s_0, \hat{\sigma}_{[0, (i+1)-1]}) && \text{(Def. of concat.)} \\ &= S_{i+1} && \text{(Def. of } S_i) \end{aligned}$$

Therefore, p_{i+1} is a bound process of \hat{p}_i .

IV. Finally, to satisfy Def. 5, it must be shown that $p_0 = B(\hat{p})$ and that there exists $\hat{p}' = U(p_{|\hat{\sigma}|-1})$. As the elements of \mathbb{P} are defined, $p_0 = (\hat{\mathcal{A}}, \hat{\sigma}, 1, \mathcal{A}_0, s_0, \hat{\sigma}_0)$. Now from Definition 4, $B(\hat{p})$ with a single input string of $\hat{\sigma}_0$ is $(\hat{\mathcal{A}}, \hat{\sigma}, 1, \mathcal{A}, s_0, \hat{\sigma}_0)$, and this equals p_0 . By Theorem 2, \hat{p}' exists.

Therefore, the requirements of Def. 5 are met, and \hat{P} is realm executable in \mathcal{R} . \square

Theorem 3 serves an important role for migration systems. It states that for a state-equivalent restricted realm, if the realm as a whole supports the entire input alphabet of a process, then that process is executable to completion within the realm. For an implementation, an SE restricted realm's processors differ only in the extent to which they support the instruction set. In a subset realm, any given machine supports a subset (including the empty set) of the program's input symbols. A homogeneous subset realm is an interesting variation, but provides no value. By definition, no single machine can execute the entire program, but also, since all machines are identical, the entire realm cannot execute the entire program. Formally, it does not fulfill the requirements of Theorem 3.

Theorem 4. *A homogeneous subset realm is never realm executable.*

Proof. Theorem 3 presents the requirement for realm executability in a subset (restricted) realm, where $\hat{A} \subseteq A_1 \cup \dots \cup A_n$. For a homogeneous realm, $A_1 = A_2 = \dots = A_n$, and the requirement simplifies to $\hat{A} \subseteq A_1$, which can never be true because it is mutually exclusive with the subset realm requirement that $(A_1 \cap \hat{A}) \subsetneq \hat{A}$. \square

Let us next consider an SE superset realm. This is a realm composed of machines that all process a superset of the program's alphabet. This is the type of realm in existing implementations that is often called a homogeneous cluster. A superset realm can be either homogeneous or heterogeneous, and the properties are identical: From the perspective of the process, all machines in the realm might as well be homogeneous because they can all execute the entire program.

One final type of realm to consider is the identity realm. An identity realm is a homogeneous realm consisting of one or more FSMs that implement exactly the abstract FSM of a program. Theorem 1 proves that a program is realm executable in an identity realm consisting of a single machine. The fact that it supports the minimum required alphabet translates into the smallest processor in an implementation that can complete the program. Using the characteristics of the realms defined above, we form a taxonomy shown in Fig. 4.

During the explanation of this model, the input symbols are viewed as being analogous to instructions for a micro-

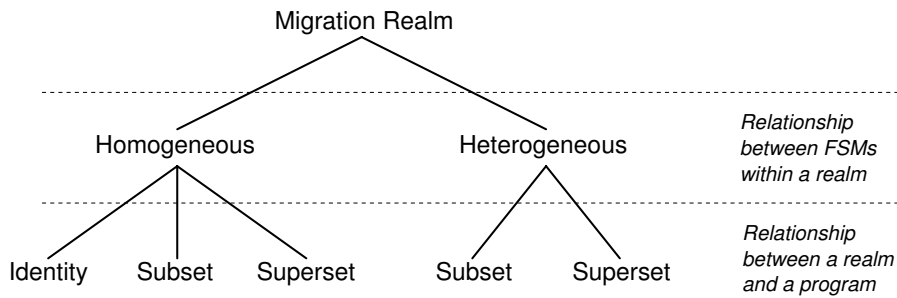


Fig. 4. Migration Realm Taxonomy

processor, but the level of abstraction is not relevant to the model. For instance, a PowerPC processor and an x86 processor are considered heterogeneous because they have different instruction sets, among other things. However, if the programs are modeled at a higher level of abstraction, then those two machines can be considered homogeneous. Two practical examples of abstraction which can make two heterogeneous processors behave in a homogeneous fashion are Remote Procedure Calls and the Java Platform.

6. APPLICATION TO HW/SW MIGRATION

In our application, the desire is to improve the performance of digital logic simulation by running the processes describing the behavior of the circuit in parallel in an FPGA. Since the parallel hardware resources are limited, they must be used efficiently by migrating idle processes out and busy processes in. The performance of a process migration system depends upon the implementation, and there are three penalties that need to be considered: instantiation overhead, migration overhead, and area utilization. The instantiation overhead is the time required to reconfigure a portion of the FPGA at run time to create or remove a processor. The migration overhead is the time to bind and unbind a process to a processor in the FPGA. The area utilization is a measure of the use of the logic structures of the FPGA. Minimizing the area utilization maximizes the number of processors.

This model shows that a completely heterogeneous realm provides no value to the execution of a process because of the problem of knowing if the result is correct. But fully heterogeneous systems are only theoretical: Computer systems all share homogeneous concepts such as add and multiply. Through requiring certain homogeneous characteristics, some of the most expensive requirements for correctness are eliminated. In contrast to full heterogeneity is full homogeneity, typically in the form of a superset realm for which a compiler ensures that the program is a subset of the supported states and alphabet. A homogeneous superset realm, however, does not maximize the use of an FPGA's area. A more efficient use is to place a process within a processor that supports just the instructions the process

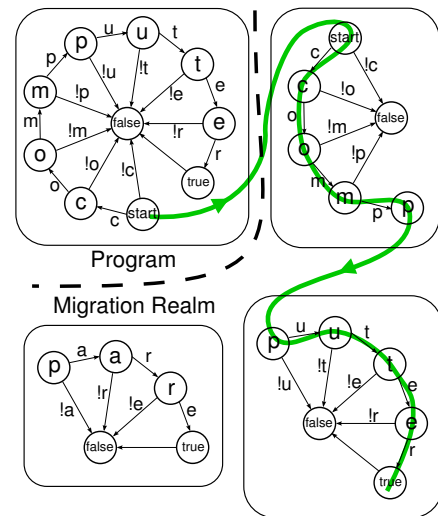


Fig. 5. An example of subset migration

utilizes—an identity realm. The instantiation overheads, however, must also be considered. If there already exists one or more processors in the FPGA that support the process's instruction set (an SE subset realm), then completion is guaranteed without incurring instantiation overhead.

Example. Continuing the example of the “computer” acceptor, Fig. 5 shows that a subset realm can provide the means to complete the recognition of “computer” while also providing recognition of “compare.” Sharing an acceptor for the common prefix “comp” reduces the area used within the FPGA compared to two separate FSMs for “computer” and “compare.” Fig. 5 shows 24 arcs in the migration realm, which would correspond directly to processing logic. Two separate FSMs would require a total of 32 arcs.

The type of realm that is used is highly dependent on the implementation and the trade-offs between performance and area utilization. Fig. 6 shows a qualitative arrangement of the three useful types of migration realms. The most efficient in migration and instantiation overhead is the superset realm because no instantiation is required, and only one migration is required for program completion. Also shown, however, is that the superset realm requires the greatest area

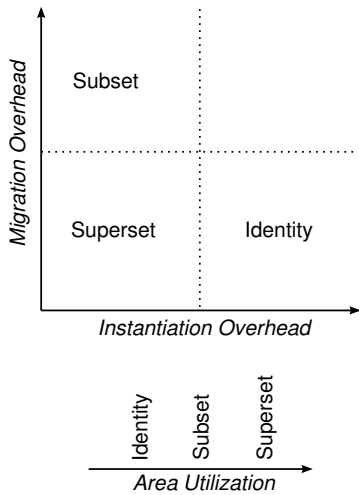


Fig. 6. The tradeoffs available for different types of realms

resources in the FPGA because it contains more than is necessary to execute the program to completion. The smallest area to support a single program is provided by the identity realm where by definition only the required instructions are supported. But an identity realm potentially requires an instantiation of a processor for every migration, thus incurring two overheads. The subset realm may require less instantiation overhead because of processor reuse, but the migration overhead increases because no single processor can execute the program to completion. Any optimal solution based on migration realms is likely to use a hybrid approach.

All the useful realms developed in this paper are state-equivalent restricted realms. Using Theorem 3, we can propose an algorithm that allows us to guarantee complete execution of a process within such a realm. Execution units in the FPGA will be configured, reconfigured, defragmented, and removed during execution, but if a list of the instructions supported by the realm at any given time is maintained, the system can check a process against the list to determine realm executability. Another method of ensuring realm executability is always to maintain one machine that is a superset machine of all the processes. Then, regardless of what kind of subsets are in the remainder of the realm, there is always this superset machine to fall back on.

Theorem 5. *An SE restricted realm of \hat{P} having one superset machine is realm executable.*

Proof. Let \mathcal{A}' be a superset machine of \hat{P} in the SE restricted realm $\mathcal{R} = \{\mathcal{A}_0, \dots, \mathcal{A}_n\}$. Since $\hat{A} \subsetneq A'$, then it follows that $\hat{A} \subseteq (A' \cup A_0 \cup \dots \cup A_n)$, satisfying Theorem 3. Therefore \hat{P} is realm executable in \mathcal{R} . \square

The latter method is the one employed in our migration system. Because of the limited FPGA resources, it is possible that all of the processes in the realm cannot fit into parallel execution units in the FPGA, and so the system main-

tains the remainder of the processes in software superset machines, thus guaranteeing realm executability regardless of the contents of the FPGA.

7. FUTURE WORK AND CONCLUSION

The model is complete enough to draw some conclusions, but future work is required. Of special interest is the development of theorems for non-state-equivalent migration realms, a more difficult problem to address. The area utilization of an FPGA can be reduced by using a subset of the state space (e.g., smaller memories), but we must still be certain of correct execution.

Furthermore, a migration realm and a program can be represented as a directed graph with FSMs as nodes and arcs as migrations. In Def. 5, realm executability is expressed in terms of a finite sequence of bound processes, and the execution path can be used to construct the arcs and nodes of a graph. The graph starts at the program's abstract FSM, and each element of the execution path forms the arc and the endpoint for each hop of the path. The labels on the arcs denote the substrings of the input string bindable to the FSM at the arc's endpoint. A path through the graph represents the executability of the process through the realm.

This paper presents a model that describes programs, processes, and the migration of those processes among a group of finite state machines. It establishes the criteria that must be met in order for the execution of the process to be correct and complete. The definition of different types of realms results in a simple migration realm taxonomy, and the theorems guide us in the development of algorithms to guarantee complete and correct execution, while still maintaining a flexibility in the utilization of the FPGA.

8. REFERENCES

- [1] D. Brand and P. Zafiropulo, "On communicating finite-state machines," *J. ACM*, vol. 30, no. 2, pp. 323–342, 1983.
- [2] F. Petrot, D. Hommais, and A. Greiner, "Cycle precise core based hardware/software system simulation with predictable event propagation," *Euromicro*, vol. 00, p. 182, 1997.
- [3] R. Milner, *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., 1982.
- [4] G.-C. Roman, P. J. McCann, and J. Y. Plun, "Mobile UNITY: reasoning and specification in mobile computing," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, pp. 250–282, 1997.
- [5] S. Lu and C. Xu, "A formal framework for agent itinerary specification, security reasoning and logic analysis," in *Dist. Computing Systems Workshops, 2005*, 2005, pp. 580–586.
- [6] D. Milojicic, F. Douglass, Y. Paidaveine, R. Wheeler, and S. Zhou., "Process migration," *ACM Computing Surveys*, vol. 32, no. 3, pp. 241–299, 2000.